# OpenShift Container Platform 4.19

## Multiple networks

Configuring and managing multiple network interfaces and virtual routing in OpenShift Container Platform

Last Updated: 2026-03-30

# OpenShift Container Platform 4.19 Multiple networks

Configuring and managing multiple network interfaces and virtual routing in OpenShift Container Platform

## Legal Notice



## Abstract

This document explains how to set up and manage primary and secondary networks, including virtual routing and forwarding, in OpenShift Container Platform.

# Table of Contents

# CHAPTER 1. UNDERSTANDING MULTIPLE NETWORKS

OpenShift Container Platform administrators and users can use user-defined networks (UDNs) or **NetworkAttachmentDefinition** (NADs) to define the networks that handle all of the ordinary network traffic of the cluster.

## 1.1. MULTIPLE NETWORKS WITH THE OVN-K CNI

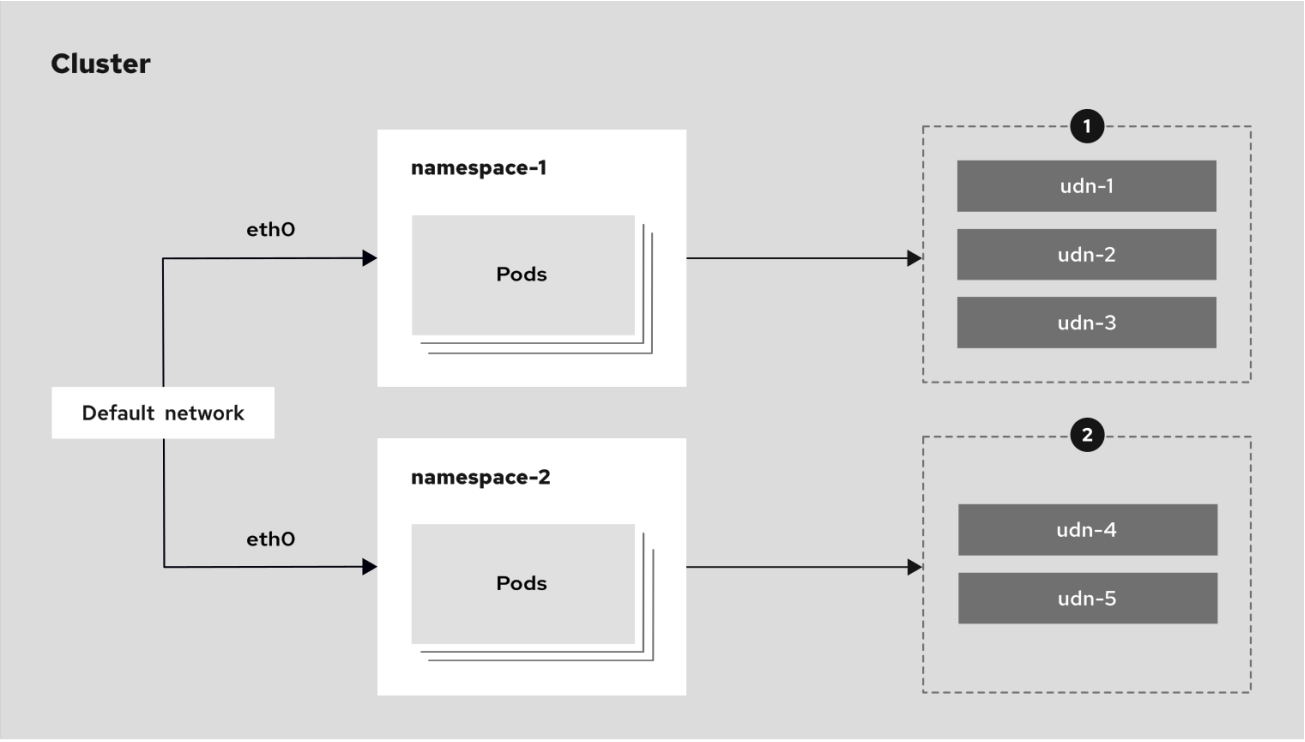
By default, OVN-Kubernetes serves as the Container Network Interface (CNI) of an OpenShift Container Platform cluster. This network interface is what administrators use to create default networks.

Both user-defined networks and Network Attachment Definitions can serve as the following network types:

- **Primary networks**: Act as the primary network for the pod. By default, all traffic passes through the primary network unless you have configured a pod route to send traffic through other networks.
- **Secondary networks**: Act as secondary, non-default networks for a pod. Secondary networks offer separate interfaces dedicated to specific traffic types or purposes. Only pod traffic that you explicitly configure to use a secondary network routes through its interface.

The following diagram shows a cluster that has an existing default network infrastructure that uses a physical network interface, **eth0**, to connect to two namespaces. Pods or virtual machines (VMs) in one namespace run in isolation from pods or VMs in the other namespace. You can create only one primary network. However, you can create multiple secondary networks for each namespace.

**Figure 1.1. Diagram showing namespaces with multiple secondary UDNs**




1. udn-1 (primary network); udn-2 and udn-3 (secondary networks)
2. udn-4 (primary network); udn-5 (secondary network)

During cluster installation, OpenShift Container Platform administrators can configure alternative

default secondary pod networks by leveraging the Multus CNI plugin. With Multus, you can use multiple CNI plugins such as ipvlan, macvlan, or Network Attachment Definitions together to serve as secondary networks for pods.

IMPORTANT

User-defined networks are only supported when OVN-Kubernetes is used as the CNI. UDNs are not supported for use with other CNIs.

You can define an secondary network based on the available CNI plugins and attach one or more of these networks to your pods. You can define more than one secondary network for your cluster depending on your needs. This gives you flexibility when you configure pods that deliver network functionality, such as switching or routing. For more information, see the links in the Additional resources:

- For a complete list of supported CNI plugins, see "Secondary networks in OpenShift Container Platform".
- For information about user-defined networks, see "About user-defined networks (UDNs)".
- For information about Network Attachment Definitions, "Creating primary networks by using a NetworkAttachmentDefinition".

## 1.2. USERDEFINEDNETWORK AND NETWORKATTACHMENTDEFINITION SUPPORT MATRIX

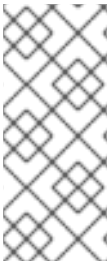
You can use user defined networks and network attachment definitions to define and configure customized networks for your needs.

By creating **UserDefinedNetwork** and **NetworkAttachmentDefinition** custom resources (CRs), cluster administrators can complete the following tasks:

- Create customizable network configurations
- Define their own network topologies
- Ensure network isolation
- Manage IP addressing for workloads
- Configure advanced network features

By creating a **ClusterUserDefinedNetwork** CR, administrators can create and define secondary networks that span multiple namespaces at the cluster level.

User-defined networks and network attachment definitions can serve as both the primary and secondary network interface, and each support **layer2** and **layer3** topologies.



NOTE

As of OpenShift Container Platform 4.19, the use of the **Localnet** topology by **ClusterUserDefinedNetwork** CRs is generally available. This configuration is the preferred method for connecting physical networks to virtual networks. Or, you can use the **NetworkAttachmentDefinition** CR to create secondary networks with **Localnet** topologies.

The following section highlights the supported features of the **UserDefinedNetwork** and **NetworkAttachmentDefinition** CRs when used as either the primary or secondary network. A separate table for the **ClusterUserDefinedNetwork** CR is also included.

**Table 1.1. Primary network support matrix for UserDefinedNetwork and NetworkAttachmentDefinition CRs**

| Network feature | Layer2 topology | Layer3 topology |
|---|---|---|
| east-west traffic | ✓ | ✓ |
| north-south traffic | ✓ | ✓ |
| Persistent IPs | ✓ | X |
| Services | ✓ | ✓ |
| Routes | X | X |
| **EgressIP** resource | ✓ | ✓ |
| Multicast | X | ✓ |
| **NetworkPolicy** resource | ✓ | ✓ |
| **MultinetworkPolicy** resource | X | X |

where:

**Multicast**

Must be enabled in the namespace, and it is only available between OVN-Kubernetes network pods. For more information, see "About multicast".

**NetworkPolicy resource**

When creating a **ClusterUserDefinedNetwork** CR with a primary network type, network policies must be created *after* the **UserDefinedNetwork** CR.

**Table 1.2. Secondary network support matrix for UserDefinedNetwork and NetworkAttachmentDefinition CRs**

| Network feature | Layer2 topology | Layer3 topology | Localnet topology |
|---|---|---|---|
| east-west traffic | ✓ | ✓ | ✓ (**NetworkAttachment Definition** CR only) |
| north-south traffic | X | X | ✓ (**NetworkAttachment Definition** CR only) |
| Persistent IPs | ✓ | X | ✓ (**NetworkAttachment Definition** CR only) |
| Services | X | X | X |
| Routes | X | X | X |
| **EgressIP** resource | X | X | X |
| Multicast | X | X | X |
| **NetworkPolicy** resource | X | X | X |
| **MultinetworkPolicy** resource | ✓ | ✓ | ✓ (**NetworkAttachment Definition** CR only) |

The Localnet topology is unavailable for use with the **UserDefinedNetwork** CR. It is only supported on secondary networks for **NetworkAttachmentDefinition** CRs.

**Table 1.3. Support matrix for ClusterUserDefinedNetwork CRs**

| Network feature | Layer2 topology | Layer3 topology | Localnet topology |
|---|---|---|---|
| east-west traffic | ✓ | ✓ | ✓ |
| north-south traffic | ✓ | ✓ | ✓ |
| Persistent IPs | ✓ | X | ✓ |
| Services | ✓ | ✓ | |
| Routes | X | X | |
| **EgressIP** resource | ✓ | ✓ | |

| Network feature | Layer2 topology | Layer3 topology | Localnet topology |
|---|---|---|---|
| Multicast | X | ✓ | |
| **MultinetworkPolicy** resource | X | X | ✓ |
| **NetworkPolicy** resource | ✓ | ✓ | |

where:

**Multicast**

must be enabled in the namespace, and it is only available between OVN-Kubernetes network pods. For more information, see "About multicast".

**NetworkPolicy resource**

When creating a **ClusterUserDefinedNetwork** CR with a primary network type, network policies must be created *after* the **UserDefinedNetwork** CR.

## 1.3. ADDITIONAL RESOURCES

- About user-defined networks (UDNs)
- Creating primary networks using a NetworkAttachmentDefinition
- Enabling multicast for a project

# CHAPTER 2. USE CASES FOR A SECONDARY NETWORK

You can use a secondary network in situations where you require network isolation, including data plane and control plane separation.

Isolating network traffic is useful for the following performance and security reasons:

- Performance
  **Traffic management**: You can send traffic on two different planes to manage how much traffic is along each plane.

- Security
  **Network isolation**: You can send sensitive traffic onto a network plane that is managed specifically for security considerations, and you can separate private data that must not be shared between tenants or customers.

All of the pods in the cluster still use the cluster-wide default network to maintain connectivity across the cluster. Every pod has an **eth0** interface that is attached to the cluster-wide pod network. You can view the interfaces for a pod by using the **oc exec -it <pod_name> -- ip a** command. If you add secondary network interfaces that use the Multus Container Network Interface (CNI). These secondary networks are named **net1**, **net2**, and so on.

To attach secondary network interfaces to a pod, you must create configurations that define how the interfaces are attached. Use either a **UserDefinedNetwork** custom resource (CR) or a **NetworkAttachmentDefinition** CR to specify each interface. A CNI configuration inside each of these CRs defines how that interface is created.

## 2.1. SECONDARY NETWORKS IN OPENSHIFT CONTAINER PLATFORM

OpenShift Container Platform provides the following CNI plugins for creating secondary networks in your cluster:

- **bridge**: To configure a bridge-based secondary network to allow pods on the same host to communicate with each other and the host, use the following procedure:
  - Configure a bridge-based secondary network

- **bond-cni**: To provide a method for aggregating multiple network interfaces into a single logical *bonded* interface, use the following procedure:
  - Configure a Bond CNI secondary network

- **host-device**: To allow pods access to a physical Ethernet network device on the host system, use the following procedure:
  - Configure a host-device secondary network

- **ipvlan**: Allow pods on a host to communicate with other hosts and pods on those hosts, similar to a macvlan-based secondary network. Unlike a macvlan-based secondary network, each pod shares the same MAC address as the parent physical network interface. Use the following procedure:
  - Configure an ipvlan-based secondary network

- **VLAN**: To allow VLAN-based network isolation and connectivity for pods, use the following procedure:

    - Configure a VLAN-based secondary network

- **macvlan**: To allow pods on a host to communicate with other hosts and pods on those hosts by using a physical network interface. Each pod that is attached to a macvlan-based secondary network is provided a unique MAC address:
    - Configure a macvlan-based secondary network
- **TAP**: A TAP device enables user space programs to send and receive network packets. To create a TAP device inside the container namespace, use the following procedure:
    - Configure a TAP-based secondary network
- **SR-IOV**: To allow pods to attach to a virtual function (VF) interface on SR-IOV capable hardware on the host system.
    - Configure an SR-IOV based secondary network
- **route-override**: To allow pods to override and set routes, use the following procedure:
    - Configure a **route-override** based secondary network

## 2.2. USERDEFINEDNETWORK AND NETWORKATTACHMENTDEFINITION SUPPORT MATRIX

You can use user defined networks and network attachment definitions to define and configure customized networks for your needs.

By creating **UserDefinedNetwork** and **NetworkAttachmentDefinition** custom resources (CRs), cluster administrators can complete the following tasks:

- Create customizable network configurations
- Define their own network topologies
- Ensure network isolation
- Manage IP addressing for workloads
- Configure advanced network features

By creating a **ClusterUserDefinedNetwork** CR, administrators can create and define secondary networks that span multiple namespaces at the cluster level.

User-defined networks and network attachment definitions can serve as both the primary and secondary network interface, and each support **layer2** and **layer3** topologies.

NOTE

As of OpenShift Container Platform 4.19, the use of the **Localnet** topology by **ClusterUserDefinedNetwork** CRs is generally available. This configuration is the preferred method for connecting physical networks to virtual networks. Or, you can use the **NetworkAttachmentDefinition** CR to create secondary networks with **Localnet** topologies.

The following section highlights the supported features of the **UserDefinedNetwork** and **NetworkAttachmentDefinition** CRs when used as either the primary or secondary network. A separate table for the **ClusterUserDefinedNetwork** CR is also included.

**Table 2.1. Primary network support matrix for UserDefinedNetwork and NetworkAttachmentDefinition CRs**

| Network feature | Layer2 topology | Layer3 topology |
|---|---|---|
| east-west traffic | ✓ | ✓ |
| north-south traffic | ✓ | ✓ |
| Persistent IPs | ✓ | X |
| Services | ✓ | ✓ |
| Routes | X | X |
| **EgressIP** resource | ✓ | ✓ |
| Multicast | X | ✓ |
| **NetworkPolicy** resource | ✓ | ✓ |
| **MultinetworkPolicy** resource | X | X |

where:

**Multicast**

Must be enabled in the namespace, and it is only available between OVN-Kubernetes network pods. For more information, see "About multicast".

**NetworkPolicy resource**

When creating a **ClusterUserDefinedNetwork** CR with a primary network type, network policies must be created *after* the **UserDefinedNetwork** CR.

**Table 2.2. Secondary network support matrix for UserDefinedNetwork and NetworkAttachmentDefinition CRs**

| Network feature | Layer2 topology | Layer3 topology | Localnet topology |
|---|---|---|---|
| east-west traffic | ✓ | ✓ | ✓ (**NetworkAttachment Definition** CR only) |
| north-south traffic | X | X | ✓ (**NetworkAttachment Definition** CR only) |

| Network feature | Layer2 topology | Layer3 topology | Localnet topology |
|---|---|---|---|
| Persistent IPs | ✓ | X | ✓ (**NetworkAttachment Definition** CR only) |
| Services | X | X | X |
| Routes | X | X | X |
| **EgressIP** resource | X | X | X |
| Multicast | X | X | X |
| **NetworkPolicy** resource | X | X | X |
| **MultinetworkPolicy** resource | ✓ | ✓ | ✓ (**NetworkAttachment Definition** CR only) |

The Localnet topology is unavailable for use with the **UserDefinedNetwork** CR. It is only supported on secondary networks for **NetworkAttachmentDefinition** CRs.

Table 2.3. Support matrix for**ClusterUserDefinedNetwork** CRs

| Network feature | Layer2 topology | Layer3 topology | Localnet topology |
|---|---|---|---|
| east-west traffic | ✓ | ✓ | ✓ |
| north-south traffic | ✓ | ✓ | ✓ |
| Persistent IPs | ✓ | X | ✓ |
| Services | ✓ | ✓ | |
| Routes | X | X | |
| **EgressIP** resource | ✓ | ✓ | |
| Multicast | X | ✓ | |
| **MultinetworkPolicy** resource | X | X | ✓ |
| **NetworkPolicy** resource | ✓ | ✓ | |

where:

**Multicast**

must be enabled in the namespace, and it is only available between OVN-Kubernetes network pods. For more information, see "About multicast".

**NetworkPolicy resource**

When creating a **ClusterUserDefinedNetwork** CR with a primary network type, network policies must be created *after* the **UserDefinedNetwork** CR.

## 2.3. ADDITIONAL RESOURCES

- Enabling multicast for a project

# CHAPTER 3. PRIMARY NETWORKS

## 3.1. ABOUT-USER-DEFINED NETWORKS

User-defined networks (UDNs) extend OVN-Kubernetes to enable custom layer 2 and layer 3 network segments with default isolation, providing enhanced network flexibility, security, and segmentation capabilities for multi-tenant deployments and custom network architectures.

### 3.1.1. Overview of user-defined networks

To secure and improve network segmentation and isolation, cluster administrators can create primary or secondary networks that span namespaces at the cluster level using the **ClusterUserDefinedNetwork** custom resource (CR) while a developer can define secondary networks at the namespace level using the **UserDefinedNetwork** CR.

Before the implementation of user-defined networks (UDN), the OVN-Kubernetes CNI plugin for OpenShift Container Platform only supported a layer 3 topology on the primary or *main* network. Due to Kubernetes design principles: all pods are attached to the main network, all pods communicate with each other by their IP addresses, and inter-pod traffic is restricted according to network policy.

While the Kubernetes design is useful for simple deployments, this layer 3 topology restricts customization of primary network segment configurations, especially for modern multi-tenant deployments.

UDN improves the flexibility and segmentation capabilities of the default layer 3 topology for a Kubernetes pod network by enabling custom layer 2 and layer 3 network segments, where all these segments are isolated by default. These segments act as either primary or secondary networks for container pods and virtual machines that use the default OVN-Kubernetes CNI plugin. UDNs enable a wide range of network architectures and topologies, enhancing network flexibility, security, and performance.

The following sections further emphasize the benefits and limitations of user-defined networks, the best practices when creating a **ClusterUserDefinedNetwork** or **UserDefinedNetwork** CR, how to create the CR, and additional configuration details that might be relevant to your deployment.

### 3.1.2. Benefits of a user-defined network

User-defined networks enable tenant isolation by providing each namespace with its own isolated primary network, reducing cross-tenant traffic risks and simplifying network management by eliminating the need for complex network policies.

User-defined networks offer the following benefits:

1. Enhanced network isolation for security
    - **Tenant isolation**: Namespaces can have their own isolated primary network, similar to how tenants are isolated in Red Hat OpenStack Platform (RHOSP). This improves security by reducing the risk of cross-tenant traffic.
2. Network flexibility
    - **Layer 2 and layer 3 support** Cluster administrators can configure primary networks as layer 2 or layer 3 network types.
3. Simplified network management

- **Reduced network configuration complexity**: With user-defined networks, the need for complex network policies are eliminated because isolation can be achieved by grouping workloads in different networks.

4. Advanced capabilities

   - **Consistent and selectable IP addressing**: Users can specify and reuse IP subnets across different namespaces and clusters, providing a consistent networking environment.

   - **Support for multiple networks** The user-defined networking feature allows administrators to connect multiple namespaces to a single network, or to create distinct networks for different sets of namespaces.

   - **Virtual machine reachability over CUDN** When you attach virtual machines (VM)s to a layer 2 **ClusterUserDefinedNetwork** with BGP route advertisements enabled, you can publish VM routes to the provider network and import routes back, avoiding per-node static routes while improving VM ingress and egress reachability.

5. Simplification of application migration from Red Hat OpenStack Platform (RHOSP)

   - **Network parity**: With user-defined networking, the migration of applications from OpenStack to OpenShift Container Platform is simplified by providing similar network isolation and configuration options.

Developers and administrators can create a user-defined network that is namespace scoped using the custom resource. An overview of the process is as follows:

1. An administrator creates a namespace for a user-defined network with the **k8s.ovn.org/primary-user-defined-network** label.

2. The **UserDefinedNetwork** CR is created by either the cluster administrator or the user.

3. The user creates pods in the namespace.

### 3.1.3. Limitations of a user-defined network

To deploy a successful user-defined networks (UDN), you must consider their limitations including DNS resolution behavior, restricted access to default network services such as the image registry, network policy constraints between isolated networks, and the requirement to create namespaces and networks before pods.

Consider the following limitations before implementing a UDN.

- **DNS limitations**:

  - DNS lookups for pods resolve to the pod's IP address on the cluster default network. Even if a pod is part of a user-defined network, DNS lookups will not resolve to the pod's IP address on that user-defined network. However, DNS lookups for services and external entities will function as expected.

  - When a pod is assigned to a primary UDN, it can access the Kubernetes API (KAPI) and DNS services on the cluster's default network.

- **Initial network assignment** You must create the namespace and network before creating pods. Assigning a namespace with pods to a new network or creating a UDN in an existing namespace will not be accepted by OVN-Kubernetes.

- **Health check limitations**: Kubelet health checks are performed by the cluster default network, which does not confirm the network connectivity of the primary interface on the pod. Consequently, scenarios where a pod appears healthy by the default network, but has broken connectivity on the primary interface, are possible with user-defined networks.

- **Network policy limitations**: Network policies that enable traffic between namespaces connected to different user-defined primary networks are not effective. These traffic policies do not take effect because there is no connectivity between these isolated networks.

- **Creation and modification limitation**: The **ClusterUserDefinedNetwork** CR and the **UserDefinedNetwork** CR cannot be modified after being created.

- **Default network service access**: A user-defined network pod is isolated from the default network, which means that most default network services are inaccessible. For example, a user-defined network pod cannot currently access the OpenShift Container Platform image registry. Because of this limitation, source-to-image builds do not work in a user-defined network namespace. Additionally, other functions do not work, including functions to create applications based on the source code in a Git repository, such as **oc new-app <command>**, and functions to create applications from an OpenShift Container Platform template that use source-to-image builds. This limitation might also affect other **openshift-*.svc** services.

- **Connectivity limitation**: NodePort services on user-defined networks are not guaranteed isolation. For example, NodePort traffic from a pod to a service on the same node is not accessible, whereas traffic from a pod on a different node succeeds.

- **Unclear error message for IP address exhaustion**: When the subnet of a user-defined network runs out of available IP addresses, new pods fail to start. When this occurs, the following error is returned: **Warning: Failed to create pod sandbox**. This error message does not clearly specify that IP depletion is the cause. To confirm the issue, you can check the **Events** page in the pod's namespace on the OpenShift Container Platform web console, where an explicit message about subnet exhaustion is reported.

- **Layer2 egress IP limitations** (**UserDefinedNetwork** CRs only):

    - Egress IP does not work without a default gateway.

    - Egress IP does not work on Google Cloud.

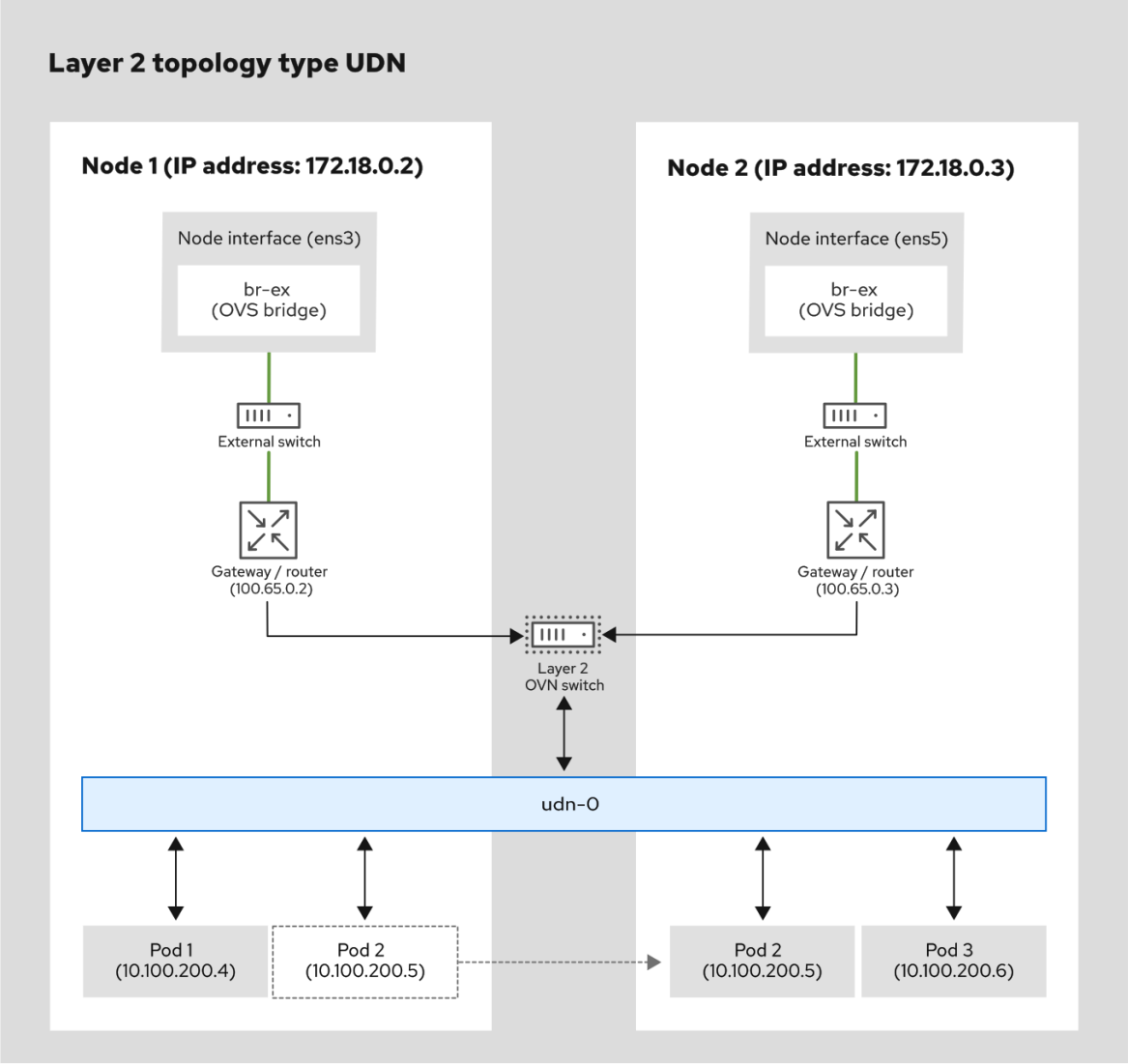
    - Egress IP does not work with multiple gateways and instead will forward all traffic to a single gateway.

### 3.1.4. Layer 2 and layer 3 topologies

A layer 2 topology creates a distributed virtual switch across cluster nodes, this network topology provides a smooth live migration of virtual machine (VM) within the same subnet. A layer 3 topology creates unique segments per node with routing between them, this network topology effectively manages large broadcast domains.

In a flat layer 2 topology, virtual machines and pods connect to the virtual switch so that all these components can communicate with each other within the same subnet. This topology is useful for the live migration of VMs across nodes in the cluster. The following diagram shows a flat layer 2 topology with two nodes that use the virtual switch for live migration purposes:

Figure 3.1. A flat layer 2 topology that uses a virtual switch for component communication



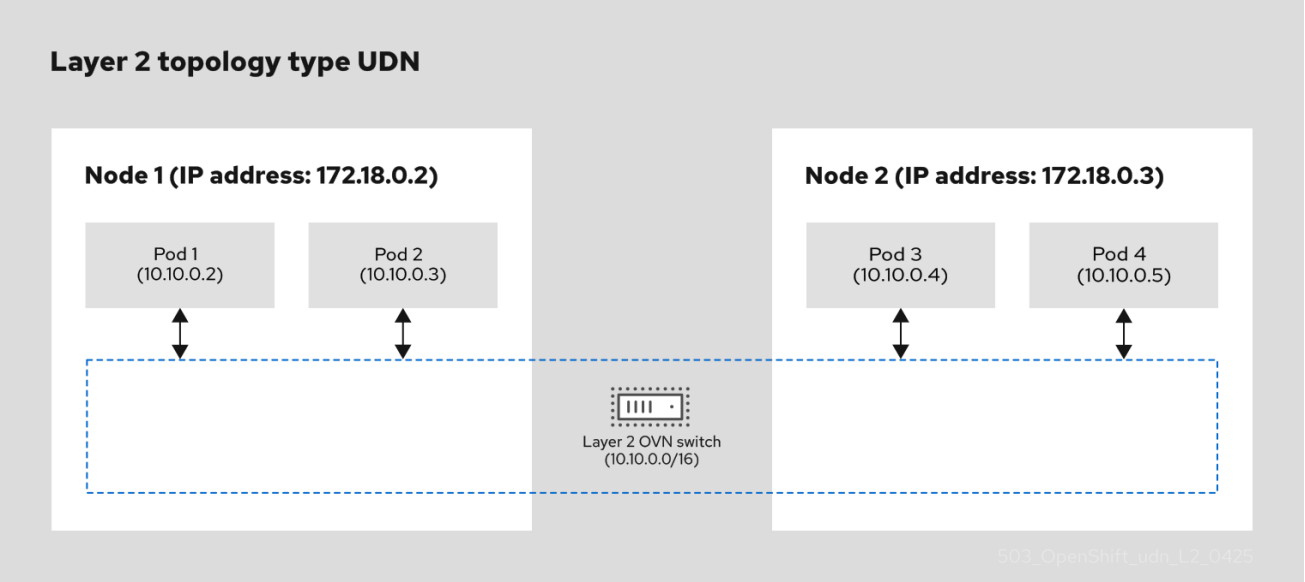

If you decide not to specify a layer 2 subnet, then you must manually configure IP addresses for each pod in your cluster. When you do not specify a layer 2 subnet, port security is limited to preventing Media Access Control (MAC) spoofing only, and does not include IP spoofing. A layer 2 topology creates a single broadcast domain that can be challenging in large network environments, where the topology might cause a broadcast storm that can degrade network performance.

To access more configurable options for your network, you can integrate a layer 2 topology with a user-defined network (UDN). The following diagram shows two nodes that use a UDN with a layer 2 topology that includes pods that exist on each node. Each node includes two interfaces:

- A node interface, which is a compute node that connects networking components to the node.
- An Open vSwitch (OVS) bridge such as **br-ex**, which creates an layer 2 OVN switch so that pods can communicate with each other and share resources.

An external switch connects these two interfaces, while the gateway or router handles routing traffic between the external switch and the layer 2 OVN switch. VMs and pods in a node can use the UDN to communicate with each other. The layer 2 OVN switch handles node traffic over a UDN so that live migrate of a VM from one node to another is possible.

Figure 3.2. A user-defined network (UDN) that uses a layer 2 topology



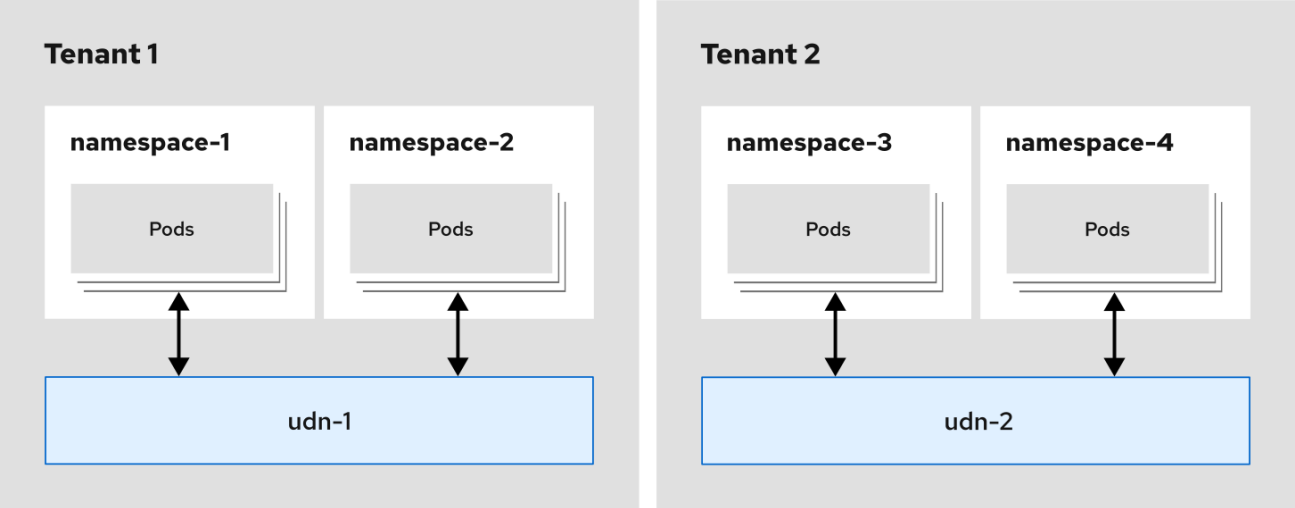

A layer 3 topology creates a unique layer 2 segment for each node in a cluster. The layer 3 routing mechanism interconnects these segments so that virtual machines and pods that are hosted on different nodes can communicate with each other. A layer 3 topology can effectively manage large broadcast domains by assigning each domain to a specific node, so that broadcast traffic has a reduced scope. To configure a layer 3 topology, you must configure **cidr** and **hostSubnet** parameters.

### 3.1.5. About the ClusterUserDefinedNetwork CR

The **ClusterUserDefinedNetwork** (CUDN) custom resource (CR) provides cluster-scoped network segmentation in OpenShift Container Platform and isolation for administrators only. Defining this resource ensures that network traffic is securely partitioned across the entire cluster.

The following diagram demonstrates how a cluster administrator can use the CUDN CR to create network isolation between tenants. This network configuration allows a network to span across many namespaces. In the diagram, network isolation is achieved through the creation of two user-defined networks, **udn-1** and **udn-2**. These networks are not connected and the **spec.namespaceSelector.matchLabels** field is used to select different namespaces. For example, **udn-1** configures and isolates communication for **namespace-1** and **namespace-2**, while **udn-2** configures and isolates communication for **namespace-3** and **namespace-4**. Isolated tenants (Tenants 1 and Tenants 2) are created by separating namespaces while also allowing pods in the same namespace to communicate.

Figure 3.3. Tenant isolation using a ClusterUserDefinedNetwork CR




### 3.1.5.1. Best practices for ClusterUserDefinedNetwork CRs

To create and deploy a successful instance of the **ClusterUserDefinedNetwork** (CUDN) CR, administrators must follow best practices such as avoiding default and openshift-* namespaces, use the proper namespace selector configuration, and ensure physical network parameter matching.

The following details provide administrators with a best practice for designing a CUDN CR:

- A **ClusterUserDefinedNetwork** CR is intended for use by cluster administrators and should not be used by non-administrators. If used incorrectly, it might result in security issues with your deployment, cause disruptions, or break the cluster network.
- **ClusterUserDefinedNetwork** CRs should not select the **default** namespace. This can result in no isolation and, as a result, could introduce security risks to the cluster.
- **ClusterUserDefinedNetwork** CRs should not select **openshift-*** namespaces.
- OpenShift Container Platform administrators should be aware that all namespaces of a cluster are selected when one of the following conditions are met:
  - The **matchLabels** selector is left empty.
  - The **matchExpressions** selector is left empty.
  - The **namespaceSelector** is initialized, but does not specify **matchExpressions** or **matchLabel**. For example: **namespaceSelector: {}**.
- For primary networks, the namespace used for the **ClusterUserDefinedNetwork** CR must include the **k8s.ovn.org/primary-user-defined-network** label. This label cannot be updated, and can only be added when the namespace is created. The following conditions apply with the **k8s.ovn.org/primary-user-defined-network** namespace label:
  - If the namespace is missing the **k8s.ovn.org/primary-user-defined-network** label and a pod is created, the pod attaches itself to the default network.

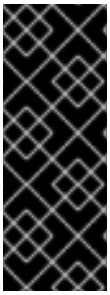
- If the namespace is missing the **k8s.ovn.org/primary-user-defined-network** label and a primary **ClusterUserDefinedNetwork** CR is created that matches the namespace, an error is reported and the network is not created.
- If the namespace is missing the **k8s.ovn.org/primary-user-defined-network** label and a primary **ClusterUserDefinedNetwork** CR already exists, a pod in the namespace is created and attached to the default network.
- If the namespace *has* the label, and a primary **ClusterUserDefinedNetwork** CR does not exist, a pod in the namespace is not created until the **ClusterUserDefinedNetwork** CR is created.

- When using the **ClusterUserDefinedNetwork** CR to create **localnet** topology, the following are best practices for administrators:
  - You must make sure that the **spec.network.physicalNetworkName** parameter matches the parameter that you configured in the Open vSwitch (OVS) bridge mapping when you create your CUDN CR. This ensures that you are bridging to the intended segment of your physical network. If you intend to deploy multiple CUDN CR using the same bridge mapping, you must ensure that the same **physicalNetworkName** parameter is used.
  - Avoid overlapping subnets between your physical network and your other network interfaces. Overlapping network subnets can cause routing conflicts and network instability. To prevent conflicts when using the **spec.network.localnet.subnets** parameter, you might use the **spec.network.localnet.excludeSubnets** parameter.
  - When you configure a Virtual Local Area Network (VLAN), you must ensure that both your underlying physical infrastructure (switches, routers, and so on) and your nodes are properly configured to accept VLAN IDs (VIDs). This means that you configure the physical network interface, for example **eth1**, as an access port for the VLAN, for example **20**, that you are connecting to through the physical switch. In addition, you must verify that an OVS bridge mapping, for example **eth1**, exists on your nodes to ensure that the physical interface is properly connected with OVN-Kubernetes.

### 3.1.5.2. Creating a ClusterUserDefinedNetwork CR by using the CLI

To implement cluster-wide network segmentation and isolation across multiple namespaces, supporting either layer 2 or layer 3 in OpenShift Container Platform, create a **ClusterUserDefinedNetwork** CR by using the CLI. Defining this resource ensures that network traffic is securely partitioned across the cluster.

Based upon your use case, create your request by using either the **cluster-layer-two-udn.yaml** example for a **Layer2** topology type or the **cluster-layer-three-udn.yaml** example for a **Layer3** topology type.



IMPORTANT

- The **ClusterUserDefinedNetwork** CR is intended for use by cluster administrators and should not be used by non-administrators. If used incorrectly, it might result in security issues with your deployment, cause disruptions, or break the cluster network.
- OpenShift Virtualization only supports the **Layer2** and **Localnet** topologies.

**Prerequisites**

- You have logged in as a user with **cluster-admin** privileges.

**Procedure**

1. Optional: For a **ClusterUserDefinedNetwork** CR that uses a primary network, create a namespace with the **k8s.ovn.org/primary-user-defined-network** label by entering the following command:

```
$ cat << EOF | oc apply -f -
apiVersion: v1
kind: Namespace
metadata:
  name: <cudn_namespace_name>
  labels:
    k8s.ovn.org/primary-user-defined-network: ""
EOF
```

2. Create a cluster-wide user-defined network for either a **Layer2** or **Layer3** topology type:

   a. Create a YAML file, such as **cluster-layer-two-udn.yaml**, to define your request for a **Layer2** topology as in the following example:

```
apiVersion: k8s.ovn.org/v1
kind: ClusterUserDefinedNetwork
metadata:
  name: <cudn_name>
spec:
  namespaceSelector:
    matchLabels:
      "<label_1_key>": "<label_1_value>"
      "<label_2_key>": "<label_2_value>"
  network:
    topology: Layer2
    layer2:
      role: Primary
      subnets:
        - "2001:db8::/64"
        - "10.100.0.0/16"
```

   where:

   **Name**

   Specifies the name of your **ClusterUserDefinedNetwork** CR.

   **namespaceSelector**

   Specifies a label query over the set of namespaces that the CUDN CR applies to. Uses the standard Kubernetes **MatchLabel** selector. Must not point to **default** or **openshift-*** namespaces.

   **matchLabels**

   Uses the **matchLabels** selector type, where terms are evaluated with an **AND** relationship. In this example, the CUDN CR is deployed to namespaces that contain both **<label_1_key>=<label_1_value>** and **<label_2_key>=<label_2_value>** labels.

   **network**

Describes the network configuration.

**topology**

This field describes the network configuration; accepted values are **Layer2** and **Layer3**. Specifying a **Layer2** topology type creates one logical switch that is shared by all nodes. This field specifies the topology configuration. It can be **layer2** or **layer3**.

**role**

Specifies **Primary** or **Secondary**. **Primary** is the only **role** specification supported in 4.19.

**subnets**

For **Layer2** topology types the following specifies config details for the field:

- The subnets field is optional.
- The subnets field is of type **string** and accepts standard CIDR formats for both IPv4 and IPv6.
- The subnets field accepts one or two items. For two items, they must be of a different family. For example, subnets values of **10.100.0.0/16** and **2001:db8::/64**.
- **Layer2** subnets can be omitted. If omitted, users must configure static IP addresses for the pods. As a consequence, port security only prevents MAC spoofing. For more information, see "Configuring pods with a static IP address".

b. Create a YAML file, such as **cluster-layer-three-udn.yaml**, to define your request for a **Layer3** topology as in the following example:

```
apiVersion: k8s.ovn.org/v1
kind: ClusterUserDefinedNetwork
metadata:
  name: <cudn_name>
spec:
  namespaceSelector:
    matchExpressions:
    - key: kubernetes.io/metadata.name
      operator: In
      values: ["<example_namespace_one>", "<example_namespace_two>"]
  network:
    topology: Layer3
    layer3:
      role: Primary
      subnets:
        - cidr: 10.100.0.0/16
          hostSubnet: 24
```

where:

**Name**

Specifies the name of your **ClusterUserDefinedNetwork** CR.

**namespaceSelector**

Specifies a label query over the set of namespaces that the CUDN CR applies to. Uses the standard Kubernetes **MatchLabel** selector. Must not point to **default** or **openshift-***

namespaces. Uses the **matchExpressions** selector type, where terms are evaluated with an **OR** relationship.

**Key**

Specifies the label key to match. Takes an operator value; valid values include: **In**, **NotIn**, **Exists**, and **DoesNotExist**. Because the **matchExpressions** type is used, provisions namespaces matching either **<example_namespace_one>** or **<example_namespace_two>**.

**network**

Describes the network configuration.

**topology**

The **topology** field describes the network configuration; accepted values are **Layer2** and **Layer3**. Specifying a **Layer3** topology type creates a layer 2 segment per node, each with a different subnet. Layer 3 routing is used to interconnect node subnets.

**role**

Specifies **Primary** or **Secondary**. **Primary** is the only **role** specification supported in 4.19.

**subnets**

For **Layer3** topology types the following specifies config details for the **subnet** field:

- The **subnets** field is mandatory.
- The type for the **subnets** field is **cidr** and **hostSubnet**:
  - **cidr** is the cluster subnet and accepts a string value.
  - **hostSubnet** specifies the nodes subnet prefix that the cluster subnet is split to.
  - For IPv6, only a **/64** length is supported for **hostSubnet**.

3. Apply your request by running the following command:

```
$ oc create --validate=true -f <example_cluster_udn>.yaml
```

Where **<example_cluster_udn>.yaml** is the name of your **Layer2** or **Layer3** configuration file.

4. Verify that your request is successful by running the following command:

```
$ oc get clusteruserdefinednetwork <cudn_name> -o yaml
```

Where **<cudn_name>** is the name you created of your cluster-wide user-defined network.

**Example output**

```
apiVersion: k8s.ovn.org/v1
kind: ClusterUserDefinedNetwork
metadata:
  creationTimestamp: "2024-12-05T15:53:00Z"
  finalizers:
  - k8s.ovn.org/user-defined-network-protection
  generation: 1
```

```
  name: my-cudn
  resourceVersion: "47985"
  uid: 16ee0fcf-74d1-4826-a6b7-25c737c1a634
spec:
  namespaceSelector:
    matchExpressions:
    - key: custom.network.selector
      operator: In
      values:
      - example-namespace-1
      - example-namespace-2
      - example-namespace-3
  network:
    layer3:
      role: Primary
      subnets:
      - cidr: 10.100.0.0/16
    topology: Layer3
status:
  conditions:
  - lastTransitionTime: "2024-11-19T16:46:34Z"
    message: 'NetworkAttachmentDefinition has been created in following namespaces:
      [example-namespace-1, example-namespace-2, example-namespace-3]'
    reason: NetworkAttachmentDefinitionReady
    status: "True"
    type: NetworkCreated
```

#### 3.1.5.3. Creating a ClusterUserDefinedNetwork CR for a Localnet topology

You deploy a **Localnet** topology to connect the secondary network to the physical underlay. This enables both east-west cluster traffic and access to services running outside the cluster. This topology type requires the additional configuration of the underlying Open vSwitch (OVS) system on cluster nodes.

**Prerequisites**

- You are logged in as a user with **cluster-admin** privileges.
- You created and configured the Open vSwitch (OVS) bridge mapping to associate the logical OVN-Kubernetes network with the physical node network through the OVS bridge. For more information, see "Configuration for a localnet switched topology".

**Procedure**

1. Create a cluster-wide user-defined network with a **Localnet** topology:
   a. Create a YAML file, such as **cluster-udn-localnet.yaml**, to define your request for a **Localnet** topology as in the following example:

```
apiVersion: k8s.ovn.org/v1
kind: ClusterUserDefinedNetwork
metadata:
  name: <cudn_name>
spec:
  namespaceSelector:
```

```
    matchLabels:
      "<label_1_key>": "<label_1_value>"
      "<label_2_key>": "<label_2_value>"
  network:
    topology: Localnet
    localnet:
      role: Secondary
      physicalNetworkName: test
      ipam: {lifecycle: Persistent}
      subnets: ["192.168.0.0/16", "2001:dbb::/64"]
```

where:

**Name**

Specifies the name of your **ClusterUserDefinedNetwork** CR.

**namespaceSelector**

Specifies a label query over the set of namespaces that the CUDN CR applies to. Uses the standard Kubernetes **MatchLabel** selector. Must not point to **default** or **openshift-*** namespaces.

**matchLabels**

Uses the **matchLabels** selector type, where terms are evaluated with an **AND** relationship. In this example, the CUDN CR is deployed to namespaces that contain both **<label_1_key>=<alabel_1_value>** and **<label_2_key>=<label_2_value>** labels.

**network**

Describes the network configuration.

**topology**

Specifying a **Localnet** topology type creates one logical switch that is directly bridged to one provider network.

**role**

Specifies the **role** for the network configuration. **Secondary** is the only **role** specification supported for the **localnet** topology.

**subnets**

For **Localnet** topology types the following specifies config details for the **subnet** field:

- The subnets field is optional.
- The subnets field is of type **string** and accepts standard CIDR formats for both IPv4 and IPv6.
- The subnets field accepts one or two items. For two items, they must be of a different IP family. For example, subnets values of **10.100.0.0/16** and **2001:db8::/64**.
- **localnet** subnets can be omitted. If omitted, users must configure static IP addresses for the pods. As a consequence, port security only prevents MAC spoofing. For more information, see "Configuring pods with a static IP address".

2. Apply your request by running the following command:

```
$ oc create --validate=true -f <example_cluster_udn>.yaml
```

where:

**<example_cluster_udn>.yaml**

Is the name of your **Localnet** configuration file.

3. Verify that your request is successful by running the following command:

```
$ oc get clusteruserdefinednetwork <cudn_name> -o yaml
```

where:

**<cudn_name>**

Is the name you created of your cluster-wide user-defined network.

**Example 3.1. Example output**

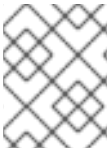
```
apiVersion: k8s.ovn.org/v1
kind: ClusterUserDefinedNetwork
metadata:
  creationTimestamp: "2025-05-28T19:30:38Z"
  finalizers:
  - k8s.ovn.org/user-defined-network-protection
  generation: 1
  name: cudn-test
  resourceVersion: "140936"
  uid: 7ff185fa-d852-4196-858a-8903b58f6890
spec:
  namespaceSelector:
    matchLabels:
      "1": "1"
      "2": "2"
  network:
    localnet:
      ipam:
        lifecycle: Persistent
      physicalNetworkName: test
      role: Secondary
      subnets:
      - 192.168.0.0/16
      - 2001:dbb::/64
    topology: Localnet
status:
  conditions:
  - lastTransitionTime: "2025-05-28T19:30:38Z"
    message: 'NetworkAttachmentDefinition has been created in following namespaces:
      [test1, test2]'
    reason: NetworkAttachmentDefinitionCreated
    status: "True"
    type: NetworkCreated
```

**Additional resources**

- Configuration for a localnet switched topology

#### 3.1.5.4. Creating a ClusterUserDefinedNetwork CR by using the web console

To implement isolated network segments with layer 2 connectivity in OpenShift Container Platform, create a **ClusterUserDefinedNetwork** custom resource (CR) by using the web console. Defining this resource ensures that your cluster workloads can communicate directly at the data link layer.



NOTE

Currently, creation of a **ClusterUserDefinedNetwork** CR with a **Layer3** topology is not supported when using the OpenShift Container Platform web console.

**Prerequisites**

- You have access to the OpenShift Container Platform web console as a user with **cluster-admin** permissions.
- You have created a namespace and applied the **k8s.ovn.org/primary-user-defined-network** label.

**Procedure**

1. From the **Administrator** perspective, click **Networking → UserDefinedNetworks**.
2. Click **ClusterUserDefinedNetwork**.
3. In the **Name** field, specify a name for the cluster-scoped UDN.
4. Specify a value in the **Subnet** field.
5. In the **Project(s) Match Labels** field, add the appropriate labels to select namespaces that the cluster UDN applies to.
6. Click **Create**. The cluster-scoped UDN serves as the default primary network for pods located in namespaces that contain the labels that you specified in step 5.

**Additional resources**

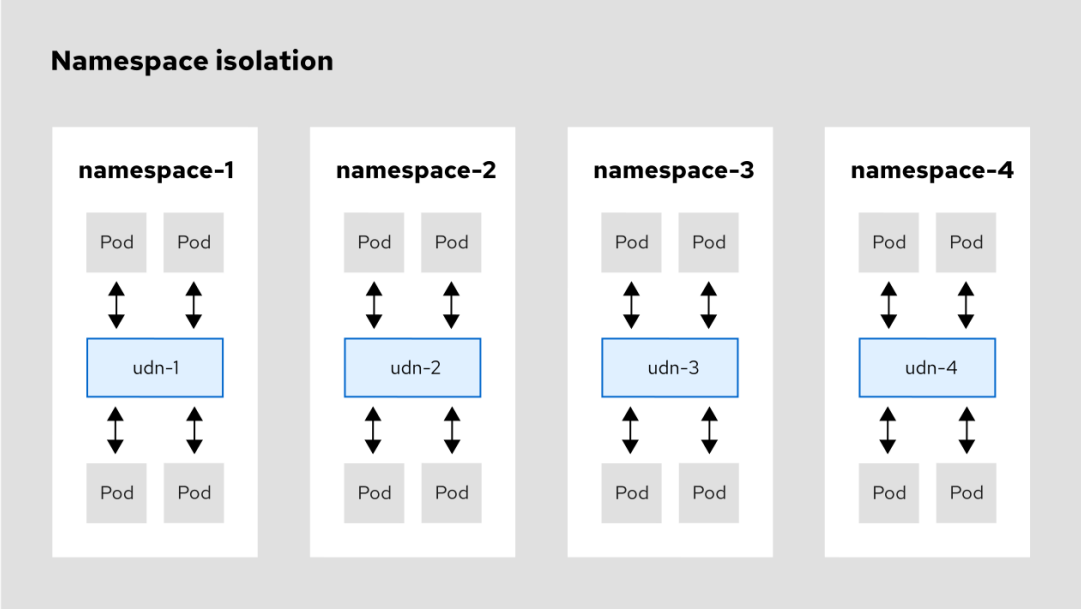
- Configuring pods with a static IP address

### 3.1.6. About the UserDefinedNetwork CR

To create advanced network segmentation and isolation, users and administrators create **UserDefinedNetwork** (UDN) custom resource (CR)s. UDNs provide granular control over network traffic within specific namespaces.

The following diagram shows four cluster namespaces, where each namespace has a single assigned user-defined network (UDN), and each UDN has an assigned custom subnet for its pod IP allocations. The OVN-Kubernetes handles any overlapping UDN subnets. Without using the Kubernetes network policy, a pod attached to a UDN can communicate with other pods in that UDN. By default, these pods are isolated from communicating with pods that exist in other UDNs. For microsegmentation, you can apply network policy within a UDN. You can assign one or more UDNs to a namespace, with a limitation of only one primary UDN to a namespace, and one or more namespaces to a UDN.

Figure 3.4. Namespace isolation using a UserDefinedNetwork CR




### 3.1.6.1. Best practices for UserDefinedNetwork CRs

To deploy a successful instance of the **UserDefinedNetwork** (UDN) CR, you must follow masquerade IP address requirements, avoid default and openshift-* namespaces, set a proper namespace selector configuration, and ensure physical network parameter matching.

The following details provide a best practice for designing a UDN CR:

- **openshift-*** namespaces should not be used to set up a **UserDefinedNetwork** CR.
- **UserDefinedNetwork** CRs should not be created in the default namespace. This can result in no isolation and, as a result, could introduce security risks to the cluster.
- For primary networks, the namespace used for the **UserDefinedNetwork** CR must include the **k8s.ovn.org/primary-user-defined-network** label. This label cannot be updated, and can only be added when the namespace is created. The following conditions apply with the **k8s.ovn.org/primary-user-defined-network** namespace label:
  - If the namespace is missing the **k8s.ovn.org/primary-user-defined-network** label and a pod is created, the pod attaches itself to the default network.
  - If the namespace is missing the **k8s.ovn.org/primary-user-defined-network** label and a primary **UserDefinedNetwork** CR is created that matches the namespace, a status error is reported and the network is not created.
  - If the namespace is missing the **k8s.ovn.org/primary-user-defined-network** label and a primary **UserDefinedNetwork** CR already exists, a pod in the namespace is created and attached to the default network.
  - If the namespace *has* the label, and a primary **UserDefinedNetwork** CR does not exist, a pod in the namespace is not created until the **UserDefinedNetwork** CR is created.
- 2 masquerade IP addresses are required for user defined networks. You must reconfigure your masquerade subnet to be large enough to hold the required number of networks.

IMPORTANT

- For OpenShift Container Platform 4.17 and later, clusters use **169.254.0.0/17** for IPv4 and **fd69::/112** for IPv6 as the default masquerade subnet. These ranges should be avoided by users. For updated clusters, there is no change to the default masquerade subnet.
- Changing the cluster's masquerade subnet is unsupported after a user-defined network has been configured for a project. Attempting to modify the masquerade subnet after a **UserDefinedNetwork** CR has been set up can disrupt the network connectivity and cause configuration issues.

- Ensure tenants are using the **UserDefinedNetwork** resource and not the **NetworkAttachmentDefinition** (NAD) CR. This can create security risks between tenants.
- When creating network segmentation, you should only use the **NetworkAttachmentDefinition** CR if user-defined network segmentation cannot be completed using the **UserDefinedNetwork** CR.
- The cluster subnet and services CIDR for a **UserDefinedNetwork** CR cannot overlap with the default cluster subnet CIDR. OVN-Kubernetes network plugin uses **100.64.0.0/16** as the default join subnet for the network. You must not use that value to configure a **UserDefinedNetwork** CR's **joinSubnets** field. If the default address values are used anywhere in the network for the cluster you must override the default values by setting the **joinSubnets** field. For more information, see "Additional configuration details for user-defined networks".

### 3.1.6.2. Creating a UserDefinedNetwork CR by using the CLI

Create a **UserDefinedNetwork** CR by using the CLI to enable namespace-scoped network segmentation and isolation, allowing you to define custom Layer 2 or Layer 3 network topologies for pods within specific namespaces.

The following procedure creates a **UserDefinedNetwork** CR that is namespace scoped. Based upon your use case, create your request by using either the **my-layer-two-udn.yaml** example for a **Layer2** topology type or the **my-layer-three-udn.yaml** example for a **Layer3** topology type.

#### Prerequisites

- As a cluster administrator, you have created a namespace.
  - During namespace creation, ensure you also applied the **k8s.ovn.org/primary-user-defined-network** label to the namespace.
  - After you create the namespace, a user that has **view** and **edit** role-based access control (RBAC) permissions can create a **UserDefinedNetwork** CR in the namespace.

#### Procedure

1. Optional: For a **UserDefinedNetwork** CR that uses a primary network, create a namespace with the **k8s.ovn.org/primary-user-defined-network** label by entering the following command:

```
$ cat << EOF | oc apply -f -
apiVersion: v1
kind: Namespace
```

```
metadata:
  name: <udn_namespace_name>
  labels:
    k8s.ovn.org/primary-user-defined-network: ""
EOF
```

2. Create a user-defined network for either a **Layer2** or **Layer3** topology type:

   a. Create a YAML file, such as **my-layer-two-udn.yaml**, to define your request for a **Layer2** topology as in the following example:

```
apiVersion: k8s.ovn.org/v1
kind: UserDefinedNetwork
metadata:
  name: udn-1
  namespace: <some_custom_namespace>
spec:
  topology: Layer2
  layer2: (1)
    role: Primary
    subnets:
      - "10.0.0.0/24"
      - "2001:db8::/60"
```

   where:

   **name**

   Name of your **UserDefinedNetwork** resource. This should not be **default** or duplicate any global namespaces created by the Cluster Network Operator (CNO).

   **topology**

   Specifies the network configuration; accepted values are **Layer2** and **Layer3**. Specifying a **Layer2** topology type creates one logical switch that is shared by all nodes.

   **role**

   Specifies a **Primary** or **Secondary** role.

   **subnets**

   For **Layer2** topology types the following specifies config details for the **subnet** field:

   - The subnets field is optional.
   - The subnets field is of type **string** and accepts standard CIDR formats for both IPv4 and IPv6.
   - The subnets field accepts one or two items. For two items, they must be of a different family. For example, subnets values of **10.100.0.0/16** and **2001:db8::/64**.
   - **Layer2** subnets can be omitted. If omitted, users must configure IP addresses for the pods. As a consequence, port security only prevents MAC spoofing.
   - The **Layer2 subnets** field is mandatory when the **ipamLifecycle** field is specified.

   b. Create a YAML file, such as **my-layer-three-udn.yaml**, to define your request for a **Layer3** topology as in the following example:

```
apiVersion: k8s.ovn.org/v1
kind: UserDefinedNetwork
metadata:
  name: udn-2-primary
  namespace: <some_custom_namespace>
spec:
  topology: Layer3
  layer3:
    role: Primary
    subnets:
      - cidr: 10.150.0.0/16
        hostSubnet: 24
      - cidr: 2001:db8::/60
        hostSubnet: 64
# ...
```

where:

**name**

Name of your **UserDefinedNetwork** resource. This should not be **default** or duplicate any global namespaces created by the Cluster Network Operator (CNO).

**topology**

Specifies the network configuration; accepted values are **Layer2** and **Layer3**. Specifying a **Layer2** topology type creates one logical switch that is shared by all nodes.

**role**

Specifies a **Primary** or **Secondary** role.

**subnets**

For **Layer3** topology types the following specifies config details for the **subnet** field:

- The **subnets** field is mandatory.
- The type for the **subnets** field is **cidr** and **hostSubnet**:
  - **cidr** is equivalent to the **clusterNetwork** configuration settings of a cluster. The IP addresses in the CIDR are distributed to pods in the user defined network. This parameter accepts a string value.
  - **hostSubnet** defines the per-node subnet prefix.
  - For IPv6, only a **/64** length is supported for **hostSubnet**.

3. Apply your request by running the following command:

```
$ oc apply -f <my_layer_two_udn>.yaml
```

Where **<my_layer_two_udn>.yaml** is the name of your **Layer2** or **Layer3** configuration file.

4. Verify that your request is successful by running the following command:

```
$ oc get userdefinednetworks udn-1 -n <some_custom_namespace> -o yaml
```

Where **some_custom_namespace** is the namespace you created for your user-defined network.

#### Example output

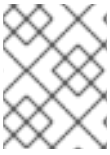
```
apiVersion: k8s.ovn.org/v1
kind: UserDefinedNetwork
metadata:
  creationTimestamp: "2024-08-28T17:18:47Z"
  finalizers:
  - k8s.ovn.org/user-defined-network-protection
  generation: 1
  name: udn-1
  namespace: some-custom-namespace
  resourceVersion: "53313"
  uid: f483626d-6846-48a1-b88e-6bbeb8bcde8c
spec:
  layer2:
    role: Primary
    subnets:
    - 10.0.0.0/24
    - 2001:db8::/60
  topology: Layer2
status:
  conditions:
  - lastTransitionTime: "2024-08-28T17:18:47Z"
    message: NetworkAttachmentDefinition has been created
    reason: NetworkAttachmentDefinitionReady
    status: "True"
    type: NetworkCreated
```

#### Additional resources

- Default cluster roles

### 3.1.6.3. Creating a UserDefinedNetwork CR by using the web console

To implement isolated network segments with layer 2 connectivity in OpenShift Container Platform, create a **UserDefinedNetwork** custom resource (CR) by using the web console. Defining this resource ensures that your cluster workloads can communicate directly at the data link layer.



NOTE

Currently, creation of a **UserDefinedNetwork** CR with a **Layer3** topology or a **Secondary** role are not supported when using the OpenShift Container Platform web console.

#### Prerequisites

- As a cluster administrator, you have created a namespace.
  - During namespace creation, ensure you also applied the **k8s.ovn.org/primary-user-defined-network** label to the namespace.

- After you create the namespace, a user that has **view** and **edit** role-based access control (RBAC) permissions can create a **UserDefinedNetwork** CR in the namespace.

**Procedure**

1. From the **Administrator** perspective, click **Networking → UserDefinedNetworks**.
2. Click **Create UserDefinedNetwork**.
3. From the **Project name** list, select the namespace that you previously created.
4. Specify a value in the **Subnet** field.
5. Click **Create**. The user-defined network serves as the default primary network for pods that you create in this namespace.

### 3.1.7. Additional configuration details for user-defined networks

Configure optional advanced settings for **ClusterUserDefinedNetwork** and **UserDefinedNetwork** CRs when default values conflict with your network topology or when you need persistent IP addresses, custom gateways, or specific subnet configurations.

It is not recommended to set these fields without explicit need and understanding of OVN-Kubernetes network topology.

1. Optional configurations for user-defined networks

| CUDN field | UDN field | Type | Description |
|---|---|---|---|

| | | | |
|---|---|---|---|
| **spec.network.<topology>.joinSubnets** | **spec.<topology>.joinSubnets** | object | When omitted, the platform sets default values for the **joinSubnets** field of **100.65.0.0/16** for IPv4 and **fd99::/64** for IPv6. If the default address values are used anywhere in the cluster's network you must override it by setting the **joinSubnets** field. If you choose to set this field, ensure it does not conflict with other subnets in the cluster such as the cluster subnet, the **default** network cluster subnet, and the masquerade subnet.<br><br>The **joinSubnets** field configures the routing between different segments within a user-defined network. Dual-stack clusters can set 2 subnets, one for each IP family; otherwise, only 1 subnet is allowed. This field is only allowed for the **Primary** network. |

| | | | |
|---|---|---|---|
| **spec.network.<topology>.excludeSubnets** | **spec.<topology>.exlcudeSubnets** | string | Specifies a list of CIDRs to be removed from the specified CIDRs in the **subnets** field. The CIDRs in this list must be in range of at least one subnet specified in **subnets**. When omitted, no IP addresses are excluded, and all IP addresses specified in the **subnets** field are subject to assignment. You must use standard CIDR notation. For example, **10.128.0.0/16**. This field must be omitted if the **subnets** field is not set or if the **ipam.mode** field is set to **Disabled**. You can only set 25 values for the **excludeSubnets** field.<br><br>When deploying a secondary network with **Localnet** topology, the IP ranges used in your physical network must be explicitly listed in the **excludeSubnets** field to prevent IP duplication in your subnet. |

| | | | |
|---|---|---|---|
| **spec.network.<topology>.ipam.lifecycle** | **spec.<topology>.ipam.lifecycle** | object | The **spec.ipam.lifecycle** field configures the IP address management system (IPAM). You might use this field for virtual workloads to ensure persistent IP addresses. The only allowed value is **Persistent**, which ensures that your virtual workloads have persistent IP addresses across reboots and migration. These are assigned by the container network interface (CNI) and used by OVN-Kubernetes to program pod IP addresses. You must not change this for pod annotations.<br><br>Setting a value of Persistent is only supported when **ipam.mode** parameter is set to **Enabled**. |

| | | | |
|---|---|---|---|
| **spec.network.<topology>.ipam.mode** | **spec.<topology>`ipam.mode** | object | The **mode** parameter controls how much of the IP configuration is managed by OVN-Kubernetes. The following options are available: * **Enabled**: When enabled, OVN-Kubernetes applies the IP configuration to the SDN infrastructure and assigns IP addresses from the selected subnet to the individual pods. This is the default setting. When set to **Enabled**, the **subnets** field must be defined. **Enabled** is the default configuration. * **Disabled**: When disabled, OVN-Kubernetes only assigns MAC addresses and provides layer 2 communication, which allows users to configure IP addresses. **Disabled** is only available for layer 2 (secondary) networks. By disabling IPAM, features that rely on selecting pods by IP, for example, network policy, services, and so on, no longer function. Additionally, IP port security is also disabled for interfaces attached to this network. The **subnets** field must be empty when **spec.ipam.mode** is set to **Disabled.** |
| **spec.network.<topology>.mtu** | **spec.<topology>.mtu** | integer | The maximum transmission units (MTU). The default value is **1400**. The boundary for IPv4 is **576**, and for IPv6 it is **1280**. |

| | | | |
|---|---|---|---|
| **spec.network.localnet.vlan** | N/A | object | This field is optional and configures the virtual local area network (VLAN) tagging and allows you to segment the physical network into multiple independent broadcast domains. |
| **spec.network.localnet.vlan.mode** | N/A | object | Acceptable values are **Access**. A value of **Access** specifies that the network interface belongs to a single VLAN and all traffic will be labelled with an **id** that is configured in the **spec.network.localnet.vlan.mode.access.id** field. The **id** specifies the VLAN **id** (VID) for access ports. Values must be an integer between 1 and 4094. |
| **spec.network.localnet.physicalNetworkName** | N/A | string | Specifies the name for a physical network interface. The value you specify must match the **network-name** parameter that you provided in your Open vSwitch (OVS) bridge mapping. |

where:

**<topology>**

Can be either **layer2** or **layer3** for the **UserDefinedNetwork** CR. For the **ClusterUserDefinedNetwork** CR the topology can also be **Localnet**.

### 3.1.8. User-defined network status condition types

To troubleshoot your network deployment in OpenShift Container Platform, evaluate the status condition types returned for **ClusterUserDefinedNetwork** and **UserDefinedNetwork** custom resources (CRs). Reviewing these conditions ensures that you can identify and resolve configuration errors.

**Table 3.1. NetworkCreated condition types (ClusterDefinedNetwork and UserDefinedNetwork CRs)**

| Condition type | Status | Reason and Message | |
|---|---|---|---|
| **NetworkCreated** | **True** | When **True**, the following reason and message is returned: | |
| | | Reason | Message |
| | | **NetworkAttachmentDefinitionCreated** | 'NetworkAttachmentDefinition has been created in following namespaces: [example-namespace-1, example-namespace-2, example-namespace-3]'` |
| **NetworkCreated** | **False** | When **False**, one of the following messages is returned: | |
| | | Reason | Message |
| | | **SyncError** | **failed to generate NetworkAttachmentDefinition** |
| | | **SyncError** | **failed to update NetworkAttachmentDefinition** |
| | | **SyncError** | **primary network already exist in namespace "<namespace_name>": "<primary_network_name>"** |
| | | **SyncError** | **failed to create NetworkAttachmentDefinition: create NAD error** |
| | | **SyncError** | **foreign NetworkAttachmentDefinition with the desired name already exist** |
| | | **SyncError** | **failed to add finalizer to UserDefinedNetwork** |
| | | **NetworkAttachmentDefinitionDeleted** | **NetworkAttachmentDefinition is being deleted: [<namespace>/<nad_name>]** |

Table 3.2. NetworkAllocationSucceeded condition types (**UserDefinedNetwork** CRs)

| Condition type | Status | Reason and Message | |
|---|---|---|---|
| **NetworkAllocationSucceeded** | **True** | When **True**, the following reason and message is returned: | |
| | | Reason | Message |

| Condition type | Status | Reason and Message | |
|---|---|---|---|
| | | **NetworkAllocationSucceeded** | **Network allocation succeeded for all synced nodes.** |
| **NetworkAllocationSucceeded** | **False** | When **False**, the following message is returned: | |
| | | Reason | Message |
| | | **InternalError** | **Network allocation failed for at least one node: [<node_name>], check UDN events for more info.** |

Table 3.3. Invalid **mtu** scenarios types for the**ClusterUserDefinedNetwork** CR

| Condition type | Reason, Message, Resolution | | |
|---|---|---|---|
| **invalid mtu** | One of the following messages is returned when the **mtu** is set incorrect: | | |
| | Reason | Message | Resolution |
| | The **mtu** field is set higher than **65536**. | **spec.network.localnet.mtu** in body should be less than **65536**. | You must set the **mtu** field lower than **65536**. |
| | The **mtu** field is set lower than **576**. | **spec.network.localnet.mtu** in body should be greater than or equal to **576**. | You must set the **mtu** field greater than or equal to **576**. |
| | The **mtu** field must be at least **1280** when using the IPv6 subnet. | **MTU should be greater than or equal to 1280 when an IPv6 subnet is used** | You must set the **mtu** field higher than or equal to **1280** when you have an IPv6 subnet defined on your user-defined network configuration. |

Table 3.4. Invalid **PhysicalNetworkName** scenarios types for the**ClusterUserDefinedNetwork** CR

| Condition type | Reason, Message, Resolution | | |
|---|---|---|---|
| **invalid PhysicalNetworkName** | One of the following messages is returned when the **PhysicalNetworkName** is set incorrect: | | |
| | Reason | Message | Resolution |

| Condition type | Reason, Message, Resolution | | |
|---|---|---|---|
| | The name of the physical network is not set. | **spec.network.localnet.physicalNetworkName: Required value** | You must set the **physicalNetworkName** field. |
| | The name of the physical network does not meet minimum length requirements. | **spec.network.localnet.physicalNetworkName in body should be at least 1 chars long** | You must set physical network name to be at least one character in length. |
| | The name of the physical network exceeds the maximum character limit of 253. | **spec.network.localnet.physicalNetworkName: Too long: may not be more than 253 bytes** | You must set physical network name to not exceed the 253 character in length. |
| | The name of the physical network must not contain **,** or **:**. | **physicalNetworkName cannot contain "," or ":" characters**. | You must remove the **,** or **:** from the physical network name. |

Table 3.5. Invalid **role** scenarios types for the**ClusterUserDefinedNetwork** CR

| Condition type | Reason, Message, Resolution | | |
|---|---|---|---|
| **role unset** or **role is primary** | One of the following messages is returned when the **spec.network.localnet.role** is set incorrect: | | |
| | Reason | Message | Resolution |
| | The **role** field must be set for your localnet topology. | **spec.network.localnet.role: Required value** | You must set the **role** field. |
| | **Primary** is not a supported value for the Localnet topology. | **spec.network.localnet.role: Unsupported value: "Primary": supported values: "Secondary"** | You must set the **role** field for your Localnet topology to **Secondary**-the accepted value. |

Table 3.6. Invalid **subnets** and **ipam** scenarios types for the**ClusterUserDefinedNetwork** CR

| Condition type | Reason, Message, Resolution | | |
|---|---|---|---|
| **LocalnetInvalidSubnets** | One of the following messages is returned when either the **spec.network.localnet.subnets** or **spec.network.localnet.ipam** is set incorrect: | | |
| | Reason | Message | Resolution |
| | The optional fields, **subnets** and **ipam.mode**, have to be set together. | **Subnets is required with ipam.mode is Enabled or unset, and forbidden otherwise** | You must set the **subnets** field unless the **spec.network.localnet.ipam.mode** is explicitly disabled. |
| | The **spec.network.localnet.subnets** must have an acceptable value when using this optional field. | **The ClusterUserDefinedNetwork "localnet-empty-subnets-fail" is invalid: spec.network.localnet.subnets: Invalid value: 0: spec.network.localnet.subnets in body should have at least 1 items** | You must set an acceptable value for **spec.network.localnet.subnets**. Acceptable values are IPv4 and IPv6 Classless Inter-Domain Routing (CIDR) ranges that do not overlap with any CIDR ranges used by OpenShift Container Platform. |
| | The **subnet** field must be set when using the optional **spec.network.localnet.excludeSubnets** field. | **excludeSubnets must be unset when subnets is unset** | You must set the **spec.network.localnet.subnets** field when using the **spec.network.localnet.excludeSubnet** field. |
| | The **excludeSubnets** must be a value within the **subnets** field. | **excludeSubnets must be subnetworks of the networks specified in the subnets field** | You must set the value for the **excludeSubnets** field to be within the **subnets** field. For example, a **subnets** value of **192.168.100.0/24** and an **excludeSubnets** value of **192.168.200.1/32** is invalid. |
| | The CIDR range is invalid. | **The ClusterUserDefinedNetwork "localnet-subnets-invalid-ipv4-cidr-fail" is invalid: spec.network.localnet.subnets[0]: Invalid value: "string": CIDR is invalid** | You must set an acceptable CIDR range for **spec.network.localnet.subnets** field. Acceptable values are IPv4 and IPv6 CIDR ranges which are not in use or reserved by OpenShift Container Platform. |

| Condition type | Reason, Message, Resolution | | |
|---|---|---|---|
| | You must set the **subnets** field when the **ipam.mode** is **Enabled** or when the IPAM mode is unset because the default value is **Enabled**. | **Subnets is required with ipam.mode is Enabled or unset, and forbidden otherwise**. | You must set the **spec.network.localnet.subnets** field unless the **spec.network.localnet.ipam.mode** is explicitly disabled. |
| | Setting two CIDR ranges for **spec.network.localnet.subnets** field requires that one be IPv4 and the other be IPv6. | **Invalid value…When 2 CIDRs are set, they must be from different IP families**. | You must change one of your CIDR ranges to a different IP family. |
| | The **spec.network.localnet.ipam.mode** is **Disabled** but the **spec.network.localnet.lifecycle** has a value of **Persistent**. | **lifecycle Persistent is only supported when ipam.mode is Enabled** | You must set the **ipam.mode** to **Enabled** when the optional field **lifecycle** has a value of **Persistent**. |

Table 3.7. Invalid **vlan** scenarios types for the**ClusterUserDefinedNetwork** CR

| Condition type | Reason, Message, Resolution | | |
|---|---|---|---|
| **invalid vlan** or **invalid mode** | One of the following messages is returned when the **spec.network.localnet.vlan** is set incorrect: | | |
| | Reason | Message | Resolution |
| | The **spec.network.localnet.vlan.mode** field must be set. | **spec.network.localnet.vlan.mode: Unsupported value: "Disabled": supported values: "Access** | You must set the **spec.network.localnet.vlan.mode** field to **Access** mode. |

| Condition type | Reason, Message, Resolution | | |
|---|---|---|---|
| | The **spec.network.localnet.vlan** field must be set when **spec.network.localnet.vlan.mode** is set to **Access** mode. | **vlan access config is required when vlan mode is 'Access', and forbidden otherwise**. | You must set **spec.network.localnet.vlan.mode.access** field when using **Access** mode. |
| | The **spec.network.localnet.vlan.access.id** value must be set when using **Access** mode. | **spec.network.localnet.vlan.access.id: Required value** | You must set a value for **spec.network.localnet.mode.access.id**. |
| | Acceptable values for **access.id** are greater than or equal to 1. | **spec.network.localnet.vlan.access.id in body should be greater than or equal to 1** | You must set a value of 1 or greater for **access.id** field. |
| | Acceptable values for **access.id** are less than or equal to 4094. | **spec.network.localnet.vlan.access.id in body should be less than or equal to 4094** | You must set a value of 4094 or less for **access.id** field. |

### 3.1.9. Opening default network ports on user-defined network pods

To allow default network pods to connect to a user-defined network pod, you can use the **k8s.ovn.org/open-default-ports** annotation. This annotation opens specific ports on the user-defined network pod for access from the default network.

By default, pods on a user-defined network (UDN) are isolated from the default network. This means that default network pods, such as those running monitoring services (Prometheus or Alertmanager) or the OpenShift Container Platform image registry, cannot initiate connections to UDN pods.

The following pod specification allows incoming TCP connections on port **80** and UDP traffic on port **53** from the default network:

```
apiVersion: v1
kind: Pod
metadata:
```

```
  annotations:
    k8s.ovn.org/open-default-ports: |
      - protocol: tcp
        port: 80
      - protocol: udp
        port: 53
# ...
```

NOTE

Open ports are accessible on the pod's default network IP, not its UDN network IP.

## 3.2. CREATING PRIMARY NETWORKS USING A NETWORKATTACHMENTDEFINITION

Use the **NetworkAttachmentDefinition** (NAD) resource to create primary networks when you need to use CNI plugins other than OVN-Kubernetes, such as IPVLAN or MACVLAN, or when you require direct control over the Container Network Interface (CNI) configuration for advanced networking scenarios.

### 3.2.1. Approaches to managing a primary network

You can manage the life cycle of a primary network created by a NAD CR through the Cluster Network Operator (CNO) or a YAML manifest. Using the CNO provides automated management of the network resource, while applying a YAML manifest allows for direct control over the network configuration.

**Modifying the Cluster Network Operator (CNO) configuration**

With this method, the CNO automatically creates and manages the **NetworkAttachmentDefinition** object. In addition to managing the object lifecycle, the CNO ensures that a DHCP is available for a primary network that uses a DHCP assigned IP address.

**Applying a YAML manifest**

With this method, you can manage the primary network directly by creating an **NetworkAttachmentDefinition** object. This approach allows for the invocation of multiple CNI plugins in order to attach primary network interfaces in a pod.

Each approach is mutually exclusive and you can only use one approach for managing a primary network at a time. For either approach, the primary network is managed by a Container Network Interface (CNI) plugin that you configure.

NOTE

When deploying OpenShift Container Platform nodes with multiple network interfaces on Red Hat OpenStack Platform (RHOSP) with OVN SDN, DNS configuration of the secondary interface might take precedence over the DNS configuration of the primary interface. In this case, remove the DNS nameservers for the subnet ID that is attached to the secondary interface by running the following command:

```
$ openstack subnet set --dns-nameserver 0.0.0.0 <subnet_id>
```

### 3.2.2. Creating a primary network attachment with the Cluster Network Operator

When you specify a primary network to create by using the Cluster Network Operator (CNO), the (CNO) creates the **NetworkAttachmentDefinition** custom resource definition (CRD) automatically and manages it.

IMPORTANT

Do not edit the **NetworkAttachmentDefinition** CRDs that the Cluster Network Operator manages. Doing so might disrupt network traffic on your primary network.

**Prerequisites**

- Install the OpenShift CLI (**oc**).
- Log in as a user with **cluster-admin** privileges.

**Procedure**

1. Optional: Create the namespace for the primary networks:

```
$ oc create namespace <namespace_name>
```

2. To edit the CNO configuration, enter the following command:

```
$ oc edit networks.operator.openshift.io cluster
```

3. Modify the CR that you are creating by adding the configuration for the primary network that you are creating, as in the following example CR.

```
apiVersion: operator.openshift.io/v1
kind: Network
metadata:
  name: cluster
spec:
  # ...
  additionalNetworks:
  - name: tertiary-net
    namespace: namespace2
    type: Raw
    rawCNIConfig: |-
      {
        "cniVersion": "0.3.1",
        "name": "tertiary-net",
        "type": "ipvlan",
        "master": "eth1",
        "mode": "l2",
        "ipam": {
          "type": "static",
          "addresses": [
            {
              "address": "192.168.1.23/24"
            }
          ]
        }
      }
```

4. Save your changes and quit the text editor to commit your changes.

**Verification**

- Confirm that the CNO created the **NetworkAttachmentDefinition** CRD by running the following command. A delay might exist before the CNO creates the CRD. The expected output shows the name of the NAD CRD and the creation age in minutes.

```
$ oc get network-attachment-definitions -n <namespace>
```

where:

**<namespace>**

Specifies the namespace for the network attachment that you added to the CNO configuration.

### 3.2.3. Configuration for a primary network attachment

You configure a primary network by using the **NetworkAttachmentDefinition** API in the **k8s.cni.cncf.io** API group.

The configuration for the API is described in the following table:

**Table 3.8. NetworkAttachmentDefinition API fields**

| Field | Type | Description |
|---|---|---|
| **metadata.name** | **string** | The name for the primary network. |
| **metadata.namespace** | **string** | The namespace that the object is associated with. |
| **spec.config** | **string** | The CNI plugin configuration in JSON format. |

### 3.2.4. Creating a primary network attachment by applying a YAML manifest

Create a primary network attachment by directly applying a **NetworkAttachmentDefinition** YAML manifest. This gives you full control over the network configuration without relying on the Cluster Network Operator to manage the resource automatically.

**Prerequisites**

- You have installed the OpenShift CLI (**oc**).
- You have logged in as a user with **cluster-admin** privileges.
- You are working in the namespace where the NAD is to be deployed.

**Procedure**

1. Create a YAML file with your primary network configuration, such as in the following example:

```
apiVersion: k8s.cni.cncf.io/v1
```

```
kind: NetworkAttachmentDefinition
metadata:
  name: next-net
spec:
  config: |-
    {
      "cniVersion": "0.3.1",
      "name": "work-network",
      "namespace": "namespace2",
      "type": "host-device",
      "device": "eth1",
      "ipam": {
        "type": "dhcp"
      }
    }
```

a. Optional: You can specify a namespace to which the NAD is applied. If you are working in the namespace where the NAD is to be deployed, the **namespace** specification is not necessary.

2. To create the primary network, enter the following command:

```
$ oc apply -f <file>.yaml
```

where:

**<file>**

Specifies the name of the file contained the YAML manifest.

# CHAPTER 4. SECONDARY NETWORKS

## 4.1. CREATING SECONDARY NETWORKS ON OVN-KUBERNETES

As a cluster administrator, you can configure a secondary network for your cluster by using the **NetworkAttachmentDefinition** (NAD) resource.

### 4.1.1. Configuration for an OVN-Kubernetes secondary network

The Red Hat OpenShift Networking OVN-Kubernetes network plugin allows the configuration of secondary network interfaces for pods. To configure secondary network interfaces, you must define the configurations in the **NetworkAttachmentDefinition** custom resource definition (CRD).

NOTE

Pod and multi-network policy creation might remain in a pending state until the OVN-Kubernetes control plane agent in the nodes processes the associated **network-attachment-definition** CRD.

You can configure an OVN-Kubernetes secondary network in layer 2, layer 3, or localnet topologies. For more information about features supported on these topologies, see "UserDefinedNetwork and NetworkAttachmentDefinition support matrix".

The following sections provide example configurations for each of the topologies that OVN-Kubernetes currently allows for secondary networks.

NOTE

Networks names must be unique. For example, creating multiple **NetworkAttachmentDefinition** CRDs with different configurations that reference the same network is unsupported.

#### 4.1.1.1. Supported platforms for OVN-Kubernetes secondary network

You can use an OVN-Kubernetes secondary network with the following supported platforms:

- Bare metal
- IBM Power®
- IBM Z®
- IBM® LinuxONE
- VMware vSphere
- Red Hat OpenStack Platform (RHOSP)

#### 4.1.1.2. OVN-Kubernetes network plugin JSON configuration table

The OVN-Kubernetes network plugin JSON configuration object describes the configuration parameters for the OVN-Kubernetes CNI network plugin. The following table details these parameters:

Table 4.1. OVN-Kubernetes network plugin JSON configuration table

| Field | Type | Description |
|---|---|---|
| **cniVersion** | **string** | The CNI specification version. The required value is **0.3.1**. |
| **name** | **string** | The name of the network. These networks are not namespaced. For example, a network named **l2-network** can be referenced by **NetworkAttachmentDefinition** custom resources (CRs) that exist in different namespaces. This configuration allows pods that use the **NetworkAttachmentDefinition** CR in different namespaces to communicate over the same secondary network. However, the **NetworkAttachmentDefinition** CRs must share the same network-specific parameters, such as **topology**, **subnets**, **mtu**, **excludeSubnets**, and **vlanID**. The **vlanID** parameter applies only when the **topology** field is set to **localnet**. |
| **type** | **string** | The name of the CNI plugin to configure. This value must be set to **ovn-k8s-cni-overlay**. |
| **topology** | **string** | The topological configuration for the network. Must be one of **layer2** or **localnet**. |
| **subnets** | **string** | The subnet to use for the network across the cluster.<br><br>For **"topology":"layer2"** deployments, IPv6 (**2001:DBB::/64**) and dual-stack (**192.168.100.0/24,2001:DBB::/64**) subnets are supported.<br><br>When omitted, the logical switch implementing the network only provides layer 2 communication, and users must configure IP addresses for the pods. Port security only prevents MAC spoofing. |
| **mtu** | **string** | The maximum transmission unit (MTU). If you do not set a value, the Cluster Network Operator (CNO) sets a default MTU value by calculating the difference among the underlay MTU of the primary network interface, the overlay MTU of the pod network, such as the Geneve (Generic Network Virtualization Encapsulation), and byte capacity of any enabled features, such as IPsec. |
| **netAttachDefName** | **string** | The metadata **namespace** and **name** of the network attachment definition CRD where this configuration is included. For example, if this configuration is defined in a **NetworkAttachmentDefinition** CRD in namespace **ns1** named **l2-network**, this should be set to **ns1/l2-network**. |
| **excludeSubnets** | **string** | A comma-separated list of CIDRs and IP addresses. IP addresses are removed from the assignable IP address pool and are never passed to the pods. |

| Field | Type | Description |
|---|---|---|
| **vlanID** | **integer** | If topology is set to **localnet**, the specified VLAN tag is assigned to traffic from this secondary network. The default is to not assign a VLAN tag. |
| **physicalNetworkName** | **string** | If topology is set to **localnet**, you can reuse the same physical network mapping with multiple network overlays. Specifies the name of the physical network to which the OVN overlay connects. When omitted, the default value is the **name** of the **localnet** network. To isolate the different networks, ensure that a different VLAN tag is used when sharing the same physical network between overlays. |

#### 4.1.1.3. Compatibility with multi-network policy

When defining a network policy, the network policy rules that can be used depend on whether the OVN-Kubernetes secondary network defines the **subnets** field.

The multi-network policy API, which is provided by the **MultiNetworkPolicy** custom resource definition (CRD) in the **k8s.cni.cncf.io** API group, is compatible with an OVN-Kubernetes secondary network.

Refer to the following table that details supported multi-network policy selectors that are based on a **subnets** CNI configuration:

| **subnets** field specified | Allowed multi-network policy selectors |
|---|---|
| Yes | • **podSelector** and **namespaceSelector**<br>• **ipBlock** |
| No | • **ipBlock** |

You can use the **k8s.v1.cni.cncf.io/policy-for** annotation on a **MultiNetworkPolicy** object to point to a **NetworkAttachmentDefinition** (NAD) custom resource (CR). The NAD CR defines the network to which the policy applies. The following example multi-network policy that uses a pod selector is valid only if the **subnets** field is defined in the secondary network CNI configuration for the secondary network named **blue2**:

```
apiVersion: k8s.cni.cncf.io/v1beta1
kind: MultiNetworkPolicy
metadata:
  name: allow-same-namespace
  annotations:
    k8s.v1.cni.cncf.io/policy-for: blue2 1
spec:
```

```
  podSelector:
  ingress:
  - from:
    - podSelector: {}
```

The following example uses the **ipBlock** network multi-network policy that is always valid for an OVN-Kubernetes secondary network:

**Example multi-network policy that uses an IP block selector**

```
apiVersion: k8s.cni.cncf.io/v1beta1
kind: MultiNetworkPolicy
metadata:
  name:  ingress-ipblock
  annotations:
    k8s.v1.cni.cncf.io/policy-for: default/flatl2net
spec:
  podSelector:
    matchLabels:
      name: access-control
  policyTypes:
  - Ingress
  ingress:
  - from:
    - ipBlock:
        cidr: 10.200.0.0/30
```

#### 4.1.1.4. Configuration for a localnet switched topology

The switched **localnet** topology interconnects the workloads created as Network Attachment Definitions (NADs) through a cluster-wide logical switch to a physical network.

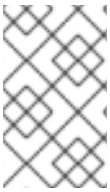
You must map a secondary network to the OVS bridge to use it as an OVN-Kubernetes secondary network. Bridge mappings allow network traffic to reach the physical network. A bridge mapping associates a physical network name, also known as an interface label, to a bridge created with Open vSwitch (OVS).

You can create an **NodeNetworkConfigurationPolicy** (NNCP) object, part of the **nmstate.io/v1** API group, to declaratively create the mapping. This API is provided by the NMState Operator. By using this API you can apply the bridge mapping to nodes that match your specified **nodeSelector** expression, such as **node-role.kubernetes.io/worker: ''**. With this declarative approach, the NMState Operator applies secondary network configuration to all nodes specified by the node selector automatically and transparently.

When attaching a secondary network, you can either use the existing **br-ex** bridge or create a new bridge. Which approach to use depends on your specific network infrastructure. Consider the following approaches:

- If your nodes include only a single network interface, you must use the existing bridge. This network interface is owned and managed by OVN-Kubernetes and you must not remove it from the **br-ex** bridge or alter the interface configuration. If you remove or alter the network interface, your cluster network stops working correctly.

- If your nodes include several network interfaces, you can attach a different network interface to a new bridge, and use that for your secondary network. This approach provides for traffic isolation from your primary cluster network.



NOTE

You cannot make configuration changes to the **br-ex** bridge or its underlying interfaces in the **NodeNetworkConfigurationPolicy** (NNCP) resource as a postinstallation task. As a workaround, use a secondary network interface connected to your host or switch.

The **localnet1** network is mapped to the **br-ex** bridge in the following sharing-a-bridge example:

```
apiVersion: nmstate.io/v1
kind: NodeNetworkConfigurationPolicy
metadata:
  name: mapping
spec:
  nodeSelector:
    node-role.kubernetes.io/worker: ''
  desiredState:
    ovn:
      bridge-mappings:
      - localnet: localnet1
        bridge: br-ex
        state: present
```

+ where:

+ **metadata.name**:: The name for the configuration object. **spec.nodeSelector.node-role.kubernetes.io/worker**:: A node selector that specifies the nodes to apply the node network configuration policy to. **spec.desiredState.ovn.bridge-mappings.localnet**:: The name for the secondary network from which traffic is forwarded to the OVS bridge. This secondary network must match the name of the **spec.config.name** field of the **NetworkAttachmentDefinition** CRD that defines the OVN-Kubernetes secondary network. **spec.desiredState.ovn.bridge-mappings.bridge**:: The name of the OVS bridge on the node. This value is required only if you specify **state: present**. **spec.desiredState.ovn.bridge-mappings.state**:: The state for the mapping. Must be either **present** to add the bridge or **absent** to remove the bridge. The default value is **present**.

+ The following JSON example configures a localnet secondary network that is named **localnet1**. Note that the value for the **mtu** parameter must match the MTU value that was set for the secondary network interface that is mapped to the **br-ex** bridge interface.

```
{
  "cniVersion": "0.3.1",
  "name": "localnet1",
  "type": "ovn-k8s-cni-overlay",
  "topology":"localnet",
  "physicalNetworkName": "localnet1",
  "subnets": "202.10.130.112/28",
  "vlanID": 33,
  "mtu": 1500,
  "netAttachDefName": "ns1/localnet-network",
  "excludeSubnets": "10.100.200.0/29"
}
```

In the following multiple interfaces example, the **localnet2** network interface is attached to the **ovs-br1** bridge. Through this attachment, the network interface is available to the OVN-Kubernetes network plugin as a secondary network.

```
apiVersion: nmstate.io/v1
kind: NodeNetworkConfigurationPolicy
metadata:
  name: ovs-br1-multiple-networks
spec:
  nodeSelector:
    node-role.kubernetes.io/worker: ''
  desiredState:
    interfaces:
    - name: ovs-br1
      description: |-
        A dedicated OVS bridge with eth1 as a port
        allowing all VLANs and untagged traffic
      type: ovs-bridge
      state: up
      bridge:
        allow-extra-patch-ports: true
        options:
          stp: false
          mcast-snooping-enable: true
        port:
        - name: eth1
    ovn:
      bridge-mappings:
      - localnet: localnet2
        bridge: ovs-br1
        state: present
```

+ where:

+ **metadata.name**:: Specifies the name of the configuration object. **node-role.kubernetes.io/worker**:: Specifies a node selector that identifies the nodes to which the node network configuration policy applies. **desiredState.interfaces.name**:: Specifies a new OVS bridge that operates separately from the default bridge used by OVN-Kubernetes for cluster traffic. **options.mcast-snooping-enable**:: Specifies whether to enable multicast snooping. When enabled, multicast snooping prevents network devices from flooding multicast traffic to all network members. By default, an OVS bridge does not enable multicast snooping. The default value is **false**. **bridge.port.name**:: Specifies the network device on the host system to associate with the new OVS bridge. **ovn.bridge-mappings.localnet**:: Specifies the name of the secondary network that forwards traffic to the OVS bridge. This name must match the value of the **spec.config.name** field in the **NetworkAttachmentDefinition** CRD that defines the OVN-Kubernetes secondary network. **ovn.bridge-mappings.bridge**:: Specifies the name of the OVS bridge on the node. The value is required only when **state: present** is set. **ovn.bridge-mappings.state**:: Specifies the state of the mapping. Valid values are **present** to add the bridge or **absent** to remove the bridge. The default value is **present**.

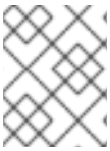
+ The following JSON example configures a localnet secondary network that is named **localnet2**. Note that the value for the **mtu** parameter must match the MTU value that was set for the **eth1** secondary network interface.

```
{
  "cniVersion": "0.3.1",
```

```
  "name": "localnet2",
  "type": "ovn-k8s-cni-overlay",
  "topology":"localnet",
  "physicalNetworkName": "localnet2",
  "subnets": "202.10.130.112/28",
  "vlanID": 33,
  "mtu": 1500,
  "netAttachDefName": "ns1/localnet-network",
  "excludeSubnets": "10.100.200.0/29"
}
```

#### 4.1.1.4.1. Configuration for a layer 2 switched topology

The switched (layer 2) topology networks interconnect the workloads through a cluster-wide logical switch. This configuration can be used for IPv6 and dual-stack deployments.



**NOTE**

Layer 2 switched topology networks only allow for the transfer of data packets between pods within a cluster.

The following JSON example configures a switched secondary network:

```
{
  "cniVersion": "0.3.1",
  "name": "l2-network",
  "type": "ovn-k8s-cni-overlay",
  "topology":"layer2",
  "subnets": "10.100.200.0/24",
  "mtu": 1300,
  "netAttachDefName": "ns1/l2-network",
  "excludeSubnets": "10.100.200.0/29"
}
```

### 4.1.1.5. Configuring pods for secondary networks

You must specify the secondary network attachments through the **k8s.v1.cni.cncf.io/networks** annotation.

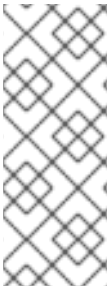
The following example provisions a pod with two secondary attachments, one for each of the attachment configurations presented in this guide:

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    k8s.v1.cni.cncf.io/networks: l2-network
  name: tinypod
  namespace: ns1
spec:
  containers:
  - args:
    - pause
```

```
    image: k8s.gcr.io/e2e-test-images/agnhost:2.36
    imagePullPolicy: IfNotPresent
    name: agnhost-container
```

#### 4.1.1.6. Configuring pods with a static IP address

You can configure pods with a static IP address. The example in the procedure provisions a pod with a static IP address.



NOTE

- You can specify the IP address for the secondary network attachment of a pod only when the secondary network attachment, a namespaced-scoped object, uses a layer 2 or localnet topology.
- Specifying a static IP address for the pod is only possible when the attachment configuration does not feature subnets.

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    k8s.v1.cni.cncf.io/networks: '[
      {
        "name": "l2-network",
        "mac": "02:03:04:05:06:07",
        "interface": "myiface1",
        "ips": [
          "192.0.2.20/24"
        ]
      }
    ]'
  name: tinypod
  namespace: ns1
spec:
  containers:
  - args:
    - pause
    image: k8s.gcr.io/e2e-test-images/agnhost:2.36
    imagePullPolicy: IfNotPresent
    name: agnhost-container
```

where:

**k8s.v1.cni.cncf.io/networks.name**

The name of the network. This value must be unique across all **NetworkAttachmentDefinition** CRDs.

**k8s.v1.cni.cncf.io/networks.mac**

The MAC address to be assigned for the interface.

**k8s.v1.cni.cncf.io/networks.interface**

The name of the network interface to be created for the pod.

**k8s.v1.cni.cncf.io/networks.ips**

The IP addresses to be assigned to the network interface.

## 4.2. CREATING SECONDARY NETWORKS WITH OTHER CNI PLUGINS

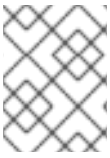
The specific configuration fields for secondary networks are described in the following sections.

### 4.2.1. Configuration for a bridge secondary network

The bridge CNI plugin JSON configuration object describes the configuration parameters for the Bridge CNI plugin. The following table details these parameters:

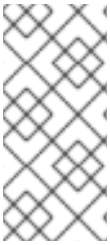
| Field | Type | Description |
|---|---|---|
| **cniVersion** | **string** | The CNI specification version. A minimum version of **0.3.1** is required. |
| **name** | **string** | The mandatory, unique identifier assigned to this CNI network attachment definition. It is used by the container runtime to select the correct network configuration and serves as the key for persistent resource state management, such as IP address allocations. |
| **type** | **string** | The name of the CNI plugin to configure: **bridge**. |
| **ipam** | **object** | The configuration object for the IPAM CNI plugin. The plugin manages IP address assignment for the attachment definition. |
| **bridge** | **string** | Optional: Specify the name of the virtual bridge to use. If the bridge interface does not exist on the host, the bridge interface gets created. The default value is **cni0**. |
| **ipMasq** | **boolean** | Optional: Set to **true** to enable IP masquerading for traffic that leaves the virtual network. The source IP address for all traffic is rewritten to the bridge's IP address. If the bridge does not have an IP address, this setting has no effect. The default value is **false**. |
| **isGateway** | **boolean** | Optional: Set to **true** to assign an IP address to the bridge. The default value is **false**. |
| **isDefaultGateway** | **boolean** | Optional: Set to **true** to configure the bridge as the default gateway for the virtual network. The default value is **false**. If **isDefaultGateway** is set to **true**, then **isGateway** is also set to **true** automatically. |
| **forceAddress** | **boolean** | Optional: Set to **true** to allow assignment of a previously assigned IP address to the virtual bridge. When set to **false**, if an IPv4 address or an IPv6 address from overlapping subsets is assigned to the virtual bridge, an error occurs. The default value is **false**. |

| Field | Type | Description |
|---|---|---|
| **hairpinMode** | **boolean** | Optional: Set to **true** to allow the virtual bridge to send an Ethernet frame back through the virtual port it was received on. This mode is also known as *reflective relay*. The default value is **false**. |
| **promiscMode** | **boolean** | Optional: Set to **true** to enable promiscuous mode on the bridge. The default value is **false**. |
| **vlan** | **string** | Optional: Specify a virtual LAN (VLAN) tag as an integer value. By default, no VLAN tag is assigned. |
| **preserveDefaultVlan** | **string** | Optional: Indicates whether the default vlan must be preserved on the **veth** end connected to the bridge. Defaults to **true**. |
| **vlanTrunk** | **list** | Optional: Assign a VLAN trunk tag. The default value is **none**. |
| **mtu** | **integer** | Optional: Set the maximum transmission unit (MTU) to the specified value. The default value is automatically set by the kernel. |
| **enabledad** | **boolean** | Optional: Enables duplicate address detection for the container side **veth**. The default value is **false**. |
| **macspoofchk** | **boolean** | Optional: Enables mac spoof check, limiting the traffic originating from the container to the mac address of the interface. The default value is **false**. |



NOTE

The VLAN parameter configures the VLAN tag on the host end of the **veth** and also enables the **vlan_filtering** feature on the bridge interface.



NOTE

To configure an uplink for an L2 network, you must allow the VLAN on the uplink interface by using the following command:

```
$ bridge vlan add vid VLAN_ID dev DEV
```

#### 4.2.1.1. Bridge CNI plugin configuration example

The following example configures a secondary network named **bridge-net**:

```
{
  "cniVersion": "0.3.1",
```

```
  "name": "bridge-net",
  "type": "bridge",
  "isGateway": true,
  "vlan": 2,
  "ipam": {
    "type": "dhcp"
    }
}
```

### 4.2.2. Configuration for a Bond CNI secondary network

The Bond Container Network Interface (Bond CNI) enables the aggregation of multiple network interfaces into a single logical bonded interface within a container, which enhances network redundancy and fault tolerance. Only SR-IOV Virtual Functions (VFs) are supported for bonding with this plugin.

The following table describes the configuration parameters for the Bond CNI plugin:

**Table 4.2. Bond CNI plugin JSON configuration object**

| Field | Type | Description |
|---|---|---|
| **name** | **string** | The mandatory, unique identifier assigned to this CNI network attachment definition. It is used by the container runtime to select the correct network configuration and serves as the key for persistent resource state management, such as IP address allocations. |
| **cniVersion** | **string** | The CNI specification version. A minimum version of **0.3.1** is required. |
| **type** | **string** | Specifies the name of the CNI plugin to configure: **bond**. |
| **miimon** | **string** | Specifies the address resolution protocol (ARP) link monitoring frequency in milliseconds. This parameter defines how often the bond interface sends ARP requests to check the availability of its aggregated interfaces. |
| **mtu** | **integer** | Optional: Specifies the maximum transmission unit (MTU) of the bond. The default is **1500**. |
| **failOverMac** | **integer** | Optional: Specifies the **failOverMac** setting for the bond. Default is **0**. |
| **mode** | **string** | Specifies the bonding policy. |
| **linksInContainer** | **boolean** | Optional: Specifies whether the network interfaces intended for bonding are expected to be created and available directly within the network namespace of the container when the bond starts. If **false** which is the default, the CNI plugin looks for these interfaces on the host system first before attempting to form the bond. |

| Field | Type | Description |
|---|---|---|
| **links** | **object** | Specifies the interfaces to be bonded. |
| **ipam** | **object** | The configuration object for the IPAM CNI plugin. The plugin manages IP address assignment for the attachment definition. |

#### 4.2.2.1. Bond CNI plugin configuration example

The following example configures a secondary network named **bond-net1**:

```
{
 "type": "bond",
 "cniVersion": "0.3.1",
 "name": "bond-net1",
 "mode": "active-backup",
 "failOverMac": 1,
 "linksInContainer": true,
 "miimon": "100",
 "mtu": 1500,
 "links": [
      {"name": "net1"},
      {"name": "net2"}
  ],
 "ipam": {
       "type": "host-local",
       "subnet": "10.56.217.0/24",
       "routes": [{
       "dst": "0.0.0.0/0"
       }],
       "gateway": "10.56.217.1"
   }
}
```

**Additional resources**

- Configuring a bond interface from two SR-IOV interfaces

### 4.2.3. Configuration for a host device secondary network

The host device CNI plugin JSON configuration object describes the configuration parameters for the host-device CNI plugin.

**NOTE**

Specify your network device by setting only one of the following parameters: **device**,**hwaddr**, **kernelpath**, or **pciBusID**.

The following table details the configuration parameters:

| Field | Type | Description |
|---|---|---|
| **cniVersion** | **string** | The CNI specification version. A minimum version of **0.3.1** is required. |
| **name** | **string** | The mandatory, unique identifier assigned to this CNI network attachment definition. It is used by the container runtime to select the correct network configuration and serves as the key for persistent resource state management, such as IP address allocations. |
| **type** | **string** | The name of the CNI plugin to configure: **host-device**. |
| **device** | **string** | Optional: The name of the device, such as **eth0**. |
| **hwaddr** | **string** | Optional: The device hardware MAC address. |
| **kernelpath** | **string** | Optional: The Linux kernel device path, such as **/sys/devices/pci0000:00/0000:00:1f.6**. |
| **pciBusID** | **string** | Optional: The PCI address of the network device, such as **0000:00:1f.6**. |

#### 4.2.3.1. host-device configuration example

The following example configures a secondary network named **hostdev-net**:

```
{
  "cniVersion": "0.3.1",
  "name": "hostdev-net",
  "type": "host-device",
  "device": "eth1"
}
```

### 4.2.4. Configuration for a dummy device additional network

The dummy CNI plugin functions like a loopback device. The plugin is a virtual interface, and you can use the plugin to route the packets to a designated IP address. Unlike a loopback device, the IP address is arbitrary and is not restricted to the **127.0.0.0/8** address range.

The dummy device CNI plugin JSON configuration object describes the configuration parameters for the dummy CNI plugin. The following table details these parameters:

| Field | Type | Description |
|---|---|---|
| **cniVersion** | **string** | The CNI specification version. A minimum version of **0.3.1** is required. |

| Field | Type | Description |
|---|---|---|
| **name** | **string** | The mandatory, unique identifier assigned to this CNI network attachment definition. It is used by the container runtime to select the correct network configuration and serves as the key for persistent resource state management, such as IP address allocations. |
| **type** | **string** | The name of the CNI plugin that you want to configure. The required value is **dummy**. |
| **ipam** | **object** | The configuration object for the IPAM CNI plugin. The plugin manages the IP address assignment for the attachment definition. |

#### 4.2.4.1. dummy configuration example

The following example configures an additional network named **hostdev-net**:

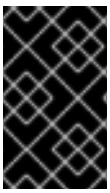
```
{
  "cniVersion": "0.3.1",
  "name": "dummy-net",
  "type": "dummy",
  "ipam": {
     "type": "host-local",
     "subnet": "10.1.1.0/24"
  }
}
```

### 4.2.5. Configuration for a VLAN secondary network

The VLAN CNI plugin JSON configuration object describes the configuration parameters for the VLAN, **vlan**, CNI plugin. The following table details these parameters:

| Field | Type | Description |
|---|---|---|
| **cniVersion** | **string** | The CNI specification version. A minimum version of **0.3.1** is required. |
| **name** | **string** | The mandatory, unique identifier assigned to this CNI network attachment definition. It is used by the container runtime to select the correct network configuration and serves as the key for persistent resource state management, such as IP address allocations. |
| **type** | **string** | The name of the CNI plugin to configure: **vlan**. |

| Field | Type | Description |
|---|---|---|
| **master** | **string** | The Ethernet interface to associate with the network attachment. If a **master** is not specified, the interface for the default network route is used. |
| **vlanId** | **integer** | Set the ID of the **vlan**. |
| **ipam** | **object** | The configuration object for the IPAM CNI plugin. The plugin manages IP address assignment for the attachment definition. |
| **mtu** | **integer** | Optional: Set the maximum transmission unit (MTU) to the specified value. The default value is automatically set by the kernel. |
| **dns** | **integer** | Optional: DNS information to return. For example, a priority-ordered list of DNS nameservers. |
| **linkInContainer** | **boolean** | Optional: Specifies whether the **master** interface is in the container network namespace or the main network namespace. Set the value to **true** to request the use of a container namespace **master** interface. |



IMPORTANT

A **NetworkAttachmentDefinition** custom resource definition (CRD) with a **vlan** configuration can be used only on a single pod in a node because the CNI plugin cannot create multiple **vlan** subinterfaces with the same **vlanId** on the same **master** interface.

#### 4.2.5.1. VLAN configuration example

The following example demonstrates a **vlan** configuration with a secondary network that is named **vlan-net**:

```
{
  "name": "vlan-net",
  "cniVersion": "0.3.1",
  "type": "vlan",
  "master": "eth0",
  "mtu": 1500,
  "vlanId": 5,
  "linkInContainer": false,
  "ipam": {
     "type": "host-local",
     "subnet": "10.1.1.0/24"
  },
  "dns": {
```

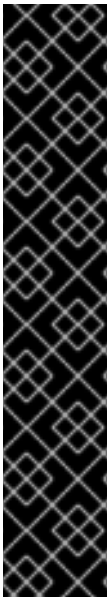
```
      "nameservers": [ "10.1.1.1", "8.8.8.8" ]
    }
  }
```

- **ipam.type.host-local**: Allocates IPv4 and IPv6 IP addresses from a specified set of address ranges. IPAM plugin stores the IP addresses locally on the host filesystem so that the addresses remain unique to the host.

### 4.2.6. Configuration for an IPVLAN secondary network

The IPVLAN CNI plugin JSON configuration object describes the configuration parameters for the IPVLAN, **ipvlan**, CNI plugin. The following table details these parameters:

| Field | Type | Description |
|---|---|---|
| **cniVersion** | **string** | The CNI specification version. A minimum version of **0.3.1** is required. |
| **name** | **string** | The mandatory, unique identifier assigned to this CNI network attachment definition. It is used by the container runtime to select the correct network configuration and serves as the key for persistent resource state management, such as IP address allocations. |
| **type** | **string** | The name of the CNI plugin to configure: **ipvlan**. |
| **ipam** | **object** | The configuration object for the IPAM CNI plugin. The plugin manages IP address assignment for the attachment definition. This is required unless the plugin is chained. |
| **mode** | **string** | Optional: The operating mode for the virtual network. The value must be **l2**, **l3**, or **l3s**. The default value is **l2**. |
| **master** | **string** | Optional: The Ethernet interface to associate with the network attachment. If a **master** is not specified, the interface for the default network route is used. |
| **mtu** | **integer** | Optional: Set the maximum transmission unit (MTU) to the specified value. The default value is automatically set by the kernel. |
| **linkInContainer** | **boolean** | Optional: Specifies whether the **master** interface is in the container network namespace or the main network namespace. Set the value to **true** to request the use of a container namespace **master** interface. |



IMPORTANT

- The **ipvlan** object does not allow virtual interfaces to communicate with the **master** interface. Therefore the container is not able to reach the host by using the **ipvlan** interface. Be sure that the container joins a network that provides connectivity to the host, such as a network supporting the Precision Time Protocol (**PTP**).
- A single **master** interface cannot simultaneously be configured to use both **macvlan** and **ipvlan**.
- For IP allocation schemes that cannot be interface agnostic, the **ipvlan** plugin can be chained with an earlier plugin that handles this logic. If the **master** is omitted, then the previous result must contain a single interface name for the **ipvlan** plugin to enslave. If **ipam** is omitted, then the previous result is used to configure the **ipvlan** interface.

#### 4.2.6.1. IPVLAN CNI plugin configuration example

The following example configures a secondary network named **ipvlan-net**:

```
{
  "cniVersion": "0.3.1",
  "name": "ipvlan-net",
  "type": "ipvlan",
  "master": "eth1",
  "linkInContainer": false,
  "mode": "l3",
  "ipam": {
    "type": "static",
    "addresses": [
      {
        "address": "192.168.10.10/24"
      }
    ]
  }
}
```

### 4.2.7. Configuration for a MACVLAN secondary network

The MACVLAN CNI plugin JSON configuration object describes the configuration parameters for the MAC Virtual LAN (MACVLAN) Container Network Interface (CNI) plugin. The following table describes these parameters:

| Field | Type | Description |
|---|---|---|
| **cniVersion** | **string** | The CNI specification version. A minimum version of **0.3.1** is required. |

| Field | Type | Description |
|---|---|---|
| **name** | **string** | The mandatory, unique identifier assigned to this CNI network attachment definition. It is used by the container runtime to select the correct network configuration and serves as the key for persistent resource state management, such as IP address allocations. |
| **type** | **string** | The name of the CNI plugin to configure: **macvlan**. |
| **ipam** | **object** | The configuration object for the IPAM CNI plugin. The plugin manages IP address assignment for the attachment definition. |
| **mode** | **string** | Optional: Configures traffic visibility on the virtual network. Must be either **bridge**, **passthru**, **private**, or **vepa**. If a value is not provided, the default value is **bridge**. |
| **master** | **string** | Optional: The host network interface to associate with the newly created macvlan interface. If a value is not specified, then the default route interface is used. |
| **mtu** | **integer** | Optional: The maximum transmission unit (MTU) to the specified value. The default value is automatically set by the kernel. |
| **linkInContainer** | **boolean** | Optional: Specifies whether the **master** interface is in the container network namespace or the main network namespace. Set the value to **true** to request the use of a container namespace **master** interface. |

NOTE

If you specify the **master** key for the plugin configuration, use a different physical network interface than the one that is associated with your primary network plugin to avoid possible conflicts.

#### 4.2.7.1. MACVLAN CNI plugin configuration example

The following example configures a secondary network named **macvlan-net**:

```
{
  "cniVersion": "0.3.1",
  "name": "macvlan-net",
  "type": "macvlan",
  "master": "eth1",
  "linkInContainer": false,
  "mode": "bridge",
  "ipam": {
    "type": "dhcp"
    }
}
```

### 4.2.8. Configuration for a TAP secondary network

The TAP CNI plugin JSON configuration object describes the configuration parameters for the TAP CNI plugin. The following table describes these parameters:

| Field | Type | Description |
|---|---|---|
| **cniVersion** | **string** | The CNI specification version. A minimum version of **0.3.1** is required. |
| **name** | **string** | The mandatory, unique identifier assigned to this CNI network attachment definition. It is used by the container runtime to select the correct network configuration and serves as the key for persistent resource state management, such as IP address allocations. |
| **type** | **string** | The name of the CNI plugin to configure: **tap**. |
| **mac** | **string** | Optional: Request the specified MAC address for the interface. |
| **mtu** | **integer** | Optional: Set the maximum transmission unit (MTU) to the specified value. The default value is automatically set by the kernel. |
| **selinuxcontext** | **string** | Optional: The SELinux context to associate with the tap device.<br>  <br>NOTE<br>The value **system_u:system_r:container_t:s0** is required for OpenShift Container Platform. |
| **multiQueue** | **boolean** | Optional: Set to **true** to enable multi-queue. |
| **owner** | **integer** | Optional: The user owning the tap device. |
| **group** | **integer** | Optional: The group owning the tap device. |
| **bridge** | **string** | Optional: Set the tap device as a port of an already existing bridge. |

#### 4.2.8.1. Tap configuration example

The following example configures a secondary network named **mynet**:

```
{
 "name": "mynet",
 "cniVersion": "0.3.1",
 "type": "tap",
```

```
 "mac": "00:11:22:33:44:55",
 "mtu": 1500,
 "selinuxcontext": "system_u:system_r:container_t:s0",
 "multiQueue": true,
 "owner": 0,
 "group": 0
 "bridge": "br1"
}
```

### 4.2.9. Setting SELinux boolean for the TAP CNI plugin

To create the tap device with the **container_t** SELinux context, enable the **container_use_devices** boolean on the host by using the Machine Config Operator (MCO).

#### Prerequisites

- You have installed the OpenShift CLI (**oc**).

#### Procedure

1. Create a new YAML file with the following details:

   **Example setsebool-container-use-devices.yaml**

```
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfig
metadata:
  labels:
    machineconfiguration.openshift.io/role: worker
  name: 99-worker-setsebool
spec:
  config:
    ignition:
      version: 3.2.0
    systemd:
      units:
      - enabled: true
        name: setsebool.service
        contents: |
          [Unit]
          Description=Set SELinux boolean for the TAP CNI plugin
          Before=kubelet.service

          [Service]
          Type=oneshot
          ExecStart=/usr/sbin/setsebool container_use_devices=on
          RemainAfterExit=true

          [Install]
          WantedBy=multi-user.target graphical.target
```

2. Create the new **MachineConfig** object by running the following command:

```
$ oc apply -f setsebool-container-use-devices.yaml
```

NOTE

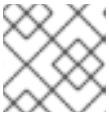
Applying any changes to the **MachineConfig** object causes all affected nodes to gracefully reboot after the change is applied. The MCO might take some time to apply the update.

Verification

- Verify that the change is applied by running the following command:

```
$ oc get machineconfigpools
```

```
NAME       CONFIG                                             UPDATED   UPDATING   DEGRADED
MACHINECOUNT   READYMACHINECOUNT   UPDATEDMACHINECOUNT
DEGRADEDMACHINECOUNT   AGE
master     rendered-master-e5e0c8e8be9194e7c5a882e047379cfa     True      False
False      3              3                 3                   0                   7d2h
worker     rendered-worker-d6c9ca107fba6cd76cdcbfcedcafa0f2     True      False      False
3              3                 3                   0                   7d
```



NOTE

All nodes should be in the **Updated** and **Ready** state.

### 4.2.10. Configuring routes using the route-override plugin on a secondary network

The Route override CNI plugin JSON configuration object describes the configuration parameters for the **route-override** CNI plugin. The following table details these parameters:

| Field | Type | Description |
|---|---|---|
| **type** | **string** | The name of the CNI plugin to configure: **route-override**. |
| **flushroutes** | **boolean** | Optional: Set to **true** to flush any existing routes. |
| **flushgateway** | **boolean** | Optional: Set to **true** to flush the default route namely the gateway route. |
| **delroutes** | **object** | Optional: Specify the list of routes to delete from the container namespace. |
| **addroutes** | **object** | Optional: Specify the list of routes to add to the container namespace. Each route is a dictionary with **dst** and optional **gw** fields. If **gw** is omitted, the plugin uses the default gateway value. |

| Field | Type | Description |
| --- | --- | --- |
| **skipcheck** | **boolean** | Optional: Set this to **true** to skip the check command. By default, CNI plugins verify the network setup during the container lifecycle. When modifying routes dynamically with **route-override**, skipping this check ensures the final configuration reflects the updated routes. |

#### 4.2.10.1. Route-override plugin configuration example

The **route-override** CNI is a type of CNI that is designed to be used when chained with a parent CNI. The CNI type does not operate independently, but relies on the parent CNI to first create the network interface and assign IP addresses before the CNI type can modify the routing rules.

The following example configures a secondary network named **mymacvlan**. The parent CNI creates a network interface attached to **eth1** and assigns an IP address in the **192.168.1.0/24** range by using **host-local** IPAM. The **route-override** CNI is then chained to the parent CNI and modifies the routing rules by flushing existing routes, deleting the route to **192.168.0.0/24**, and adding a new route for **192.168.0.0/24** with a custom gateway.

```
{
   "cniVersion": "0.3.0",
   "name": "mymacvlan",
   "plugins": [
      {
         "type": "macvlan",
         "master": "eth1",
         "mode": "bridge",
         "ipam": {
            "type": "host-local",
            "subnet": "192.168.1.0/24"
         }
      },
      {
         "type": "route-override",
         "flushroutes": true,
         "delroutes": [
            {
               "dst": "192.168.0.0/24"
            }
         ],
         "addroutes": [
            {
               "dst": "192.168.0.0/24",
               "gw": "10.1.254.254"
            }
         ]
      }
   ]
}
```

where:

**"type": "macvlan"**

The parent CNI creates a network interface attached to **eth1**.

**"type": "route-override"**

The chained **route-override** CNI modifies the routing rules.

**Additional resources**

- Setting SELinux booleans

## 4.3. ATTACHING A POD TO A SECONDARY NETWORK

To enable a pod to use additional network interfaces beyond the primary cluster network in OpenShift Container Platform, you can attach the pod to a secondary network. Secondary networks provide additional connectivity options for your workloads.

### 4.3.1. Adding a pod to a secondary network

To enable a pod to use additional network interfaces in OpenShift Container Platform, you can attach the pod to a secondary network. The pod continues to send normal cluster-related network traffic over the default network.

When a pod is created, a secondary network is attached to the pod. However, if a pod already exists, you cannot attach a secondary network to it.

The pod must be in the same namespace as the secondary network.

**Prerequisites**

- Install the OpenShift CLI (**oc**).
- Log in to the cluster.

**Procedure**

1. Add an annotation to the **Pod** object. Only one of the following annotation formats can be used:

   a. To attach a secondary network without any customization, add an annotation with the following format:

   ```
   metadata:
     annotations:
       k8s.v1.cni.cncf.io/networks: <network>[,<network>,...]
   ```

   where:

   **k8s.v1.cni.cncf.io/networks**

   Specifies the name of the secondary network to associate with the pod. To specify more than one secondary network, separate each network with a comma. Do not include whitespace between the comma. If you specify the same secondary network multiple times, that pod will have multiple network interfaces attached to that network.

   b. To attach a secondary network with customizations, add an annotation with the following format:

```
metadata:
  annotations:
    k8s.v1.cni.cncf.io/networks: |-
      [
        {
          "name": "<network>",
          "namespace": "<namespace>",
          "default-route": ["<default_route>"]
        }
      ]
```

where:

**<network>**

Specifies the name of the secondary network defined by a **NetworkAttachmentDefinition** object.

**<namespace>**

Specifies the namespace where the **NetworkAttachmentDefinition** object is defined.

**<default-route>**

Optional parameter. Specifies an override for the default route, such as **192.168.17.1**.

2. Create the pod by entering the following command.

```
$ oc create -f <name>.yaml
```

Replace **<name>** with the name of the pod.

3. Optional: Confirm that the annotation exists in the **pod** CR by entering the following command. Replace **<name>** with the name of the pod.

```
$ oc get pod <name> -o yaml
```

In the following example, the **example-pod** pod is attached to the **net1** secondary network:

```
$ oc get pod example-pod -o yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
    k8s.v1.cni.cncf.io/networks: macvlan-bridge
    k8s.v1.cni.cncf.io/network-status: |-
      [{
          "name": "ovn-kubernetes",
          "interface": "eth0",
          "ips": [
              "10.128.2.14"
          ],
          "default": true,
          "dns": {}
      },{
          "name": "macvlan-bridge",
          "interface": "net1",
          "ips": [
```

```
            "20.2.2.100"
          ],
          "mac": "22:2f:60:a5:f8:00",
          "dns": {}
      }]
  name: example-pod
  namespace: default
spec:
  ...
status:
  ...
```

where:

**k8s.v1.cni.cncf.io/network-status**

Specifies a JSON array of objects. Each object describes the status of a secondary network attached to the pod. The annotation value is stored as a plain text value.

#### 4.3.1.1. Specifying pod-specific addressing and routing options

To set static IP addresses, MAC addresses, and default routes for a pod in OpenShift Container Platform, you can configure pod-specific addressing and routing options using JSON-formatted annotations. With these annotations, you can customize network behavior for individual pods on secondary networks.

**Prerequisites**

- The pod must be in the same namespace as the secondary network.
- Install the OpenShift CLI (**oc**).
- You must log in to the cluster.

**Procedure**

1. Edit the **Pod** resource definition. If you are editing an existing **Pod** resource, run the following command to edit its definition in the default editor. Replace **<name>** with the name of the **Pod** resource to edit.

```
$ oc edit pod <name>
```

2. In the **Pod** resource definition, add the **k8s.v1.cni.cncf.io/networks** parameter to the pod **metadata** mapping. The **k8s.v1.cni.cncf.io/networks** accepts a JSON string of a list of objects that reference the name of **NetworkAttachmentDefinition** custom resource (CR) names in addition to specifying additional properties.

```
metadata:
  annotations:
    k8s.v1.cni.cncf.io/networks: '[<network>[,<network>,...]]'
# ...
```

where:

**<network>**

Replace with a JSON object as shown in the following examples. The single quotes are required.

In the following example the annotation specifies which network attachment will have the default route, using the **default-route** parameter.

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
  annotations:
    k8s.v1.cni.cncf.io/networks: '[
    {
      "name": "net1"
    },
    {
      "name": "net2",
      "default-route": ["192.0.2.1"]
    }]'
spec:
  containers:
  - name: example-pod
    command: ["/bin/bash", "-c", "sleep 2000000000000"]
    image: centos/tools
```

where:

**net1, net2**

Specifies the name of the **NetworkAttachmentDefinition** resource that defines the secondary network to associate with the pod.

**192.0.2.1**

Specifies a value of a gateway for traffic to be routed over if no other routing entry is present in the routing table. If more than one **default-route** key is specified, this will cause the pod to fail to become active.

The default route will cause any traffic that is not specified in other routes to be routed to the gateway.

IMPORTANT

Setting the default route to an interface other than the default network interface for OpenShift Container Platform may cause traffic that is anticipated for pod-to-pod traffic to be routed over another interface.

To verify the routing properties of a pod, the **oc** command may be used to execute the **ip** command within a pod.

```
$ oc exec -it <pod_name> -- ip route
```

NOTE

You may also reference the pod's **k8s.v1.cni.cncf.io/network-status** to see which secondary network has been assigned the default route, by the presence of the **default-route** key in the JSON-formatted list of objects.

To set a static IP address or MAC address for a pod you can use the JSON formatted annotations. This requires you create networks that specifically allow for this functionality. This can be specified in a rawCNIConfig for the CNO.

3. Edit the CNO CR by running the following command:

```
$ oc edit networks.operator.openshift.io cluster
```

The following YAML describes the configuration parameters for the CNO:

### Cluster Network Operator YAML configuration

```
name: <name>
namespace: <namespace>
rawCNIConfig: '{
  ...
}'
type: Raw
```

where:

**name**

Specifies a name for the secondary network attachment that you are creating. The name must be unique within the specified **namespace**.

**namespace**

Specifies the namespace to create the network attachment in. If you do not specify a value, then the **default** namespace is used.

**rawCNIConfig**

Specifies the CNI plugin configuration in JSON format, which is based on the following template.

The following object describes the configuration parameters for utilizing static MAC address and IP address using the macvlan CNI plugin:

### macvlan CNI plugin JSON configuration object using static IP and MAC address

```
{
  "cniVersion": "0.3.1",
  "name": "<name>",
  "plugins": [{
     "type": "macvlan",
     "capabilities": { "ips": true },
     "master": "eth0",
     "mode": "bridge",
     "ipam": {
       "type": "static"
     }
```

```
    }, {
      "capabilities": { "mac": true },
      "type": "tuning"
    }]
}
```

where:

**name**

Specifies the name for the secondary network attachment to create. The name must be unique within the specified **namespace**.

**plugins**

Specifies an array of CNI plugin configurations. The first object specifies a macvlan plugin configuration and the second object specifies a tuning plugin configuration.

**ips**

Specifies that a request is made to enable the static IP address functionality of the CNI plugin runtime configuration capabilities.

**master**

Specifies the interface that the macvlan plugin uses.

**mac**

Specifies that a request is made to enable the static MAC address functionality of a CNI plugin.

The above network attachment can be referenced in a JSON formatted annotation, along with keys to specify which static IP and MAC address will be assigned to a given pod.

4. Edit the pod by entering the following command:

```
$ oc edit pod <name>
```

### macvlan CNI plugin JSON configuration object using static IP and MAC address

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
  annotations:
    k8s.v1.cni.cncf.io/networks: '[
      {
        "name": "<name>",
        "ips": [ "192.0.2.205/24" ],
        "mac": "CA:FE:C0:FF:EE:00"
      }
    ]'
```

where:

**metadata.name**

Specifies the name for the secondary network attachment to create. The name must be unique within the specified **namespace**.

**metadata.annotations.k8s.v1.cni.cncf.io/ips**

Specifies an IP address including the subnet mask.

**metadata.annotations.k8s.v1.cni.cncf.io/mac**

Specifies the MAC address.

NOTE

Static IP addresses and MAC addresses do not have to be used at the same time. You can use them individually, or together.

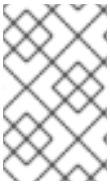
## 4.4. CONFIGURING MULTI-NETWORK POLICY

As an administrator, you can use the **MultiNetworkPolicy** API to create multiple network policies that manage traffic for pods that are attached to secondary networks. For example, you can create policies that allow or deny traffic based on specific ports, IPs and ranges, or labels.

Multi-network policies can be used to manage traffic on secondary networks in the cluster. These policies cannot manage the default cluster network or primary network of user-defined networks.

As a cluster administrator, you can configure a multi-network policy for any of the following network types:

- Single-Root I/O Virtualization (SR-IOV)
- MAC Virtual Local Area Network (MacVLAN)
- IP Virtual Local Area Network (IPVLAN)
- Bond Container Network Interface (CNI) over SR-IOV
- OVN-Kubernetes secondary networks



NOTE

Support for configuring multi-network policies for SR-IOV secondary networks is only supported with kernel network interface controllers (NICs). SR-IOV is not supported for Data Plane Development Kit (DPDK) applications.

### 4.4.1. Differences between multi-network policy and network policy

Although the **MultiNetworkPolicy** API implements the **NetworkPolicy** API, ensure that you understand the following key differences between the two policies:

- You must use the **MultiNetworkPolicy** API, as demonstrated in the following example configuration:

```
apiVersion: k8s.cni.cncf.io/v1beta1
kind: MultiNetworkPolicy
# ...
```

- You must use the **multi-networkpolicy** resource name when using the CLI to interact with multi-network policies. For example, you can view a multi-network policy object with the **oc get multi-networkpolicy <name>** command where **<name>** is the name of a multi-network policy.

- You can use the **k8s.v1.cni.cncf.io/policy-for** annotation on a **MultiNetworkPolicy** object to point to a **NetworkAttachmentDefinition** (NAD) custom resource (CR). The NAD CR defines the network to which the policy applies. The following example multi-network policy includes the **k8s.v1.cni.cncf.io/policy-for** annotation:

```
apiVersion: k8s.cni.cncf.io/v1beta1
kind: MultiNetworkPolicy
metadata:
  annotations:
    k8s.v1.cni.cncf.io/policy-for:<namespace_name>/<network_name>
# ...
```

where:

**<namespace_name>**

Specifies the namespace name.

**<network_name>**

Specifies the name of a network attachment definition.

### 4.4.2. Enabling multi-network policy for the cluster

As a cluster administrator, you can enable multi-network policy support on your cluster.

#### Prerequisites

- Install the OpenShift CLI (**oc**).
- Log in to the cluster with a user with **cluster-admin** privileges.

#### Procedure

1. Create the **multinetwork-enable-patch.yaml** file with the following YAML:

```
apiVersion: operator.openshift.io/v1
kind: Network
metadata:
  name: cluster
spec:
  useMultiNetworkPolicy: true
# ...
```

2. Configure the cluster to enable multi-network policy. Successful output lists the name of the policy object and the **patched** status.

```
$ oc patch network.operator.openshift.io cluster --type=merge --patch-file=multinetwork-enable-patch.yaml
```

### 4.4.3. Supporting multi-network policies in IPv6 networks

The ICMPv6 Neighbor Discovery Protocol (NDP) is a set of messages and processes that enable devices to discover and maintain information about neighboring nodes. NDP plays a crucial role in IPv6 networks, facilitating the interaction between devices on the same link.

The Cluster Network Operator (CNO) deploys the iptables implementation of multi-network policy when the **useMultiNetworkPolicy** parameter is set to **true**.

To support multi-network policies in IPv6 networks the Cluster Network Operator deploys the following set of custom rules in every pod affected by a multi-network policy:

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: multi-networkpolicy-custom-rules
  namespace: openshift-multus
data:

  custom-v6-rules.txt: |
    # accept NDP
    -p icmpv6 --icmpv6-type neighbor-solicitation -j ACCEPT
    -p icmpv6 --icmpv6-type neighbor-advertisement -j ACCEPT
    # accept RA/RS
    -p icmpv6 --icmpv6-type router-solicitation -j ACCEPT
    -p icmpv6 --icmpv6-type router-advertisement -j ACCEPT
```

where:

**icmpv6-type neighbor-solicitation**

This rule allows incoming ICMPv6 neighbor solicitation messages, which are part of the neighbor discovery protocol (NDP). These messages help determine the link-layer addresses of neighboring nodes.

**icmpv6-type neighbor-advertisement**

This rule allows incoming ICMPv6 neighbor advertisement messages, which are part of NDP and provide information about the link-layer address of the sender.

**icmpv6-type router-solicitation**

This rule permits incoming ICMPv6 router solicitation messages. Hosts use these messages to request router configuration information.

**icmpv6-type router-advertisement**

This rule allows incoming ICMPv6 router advertisement messages, which give configuration information to hosts.

NOTE

You cannot edit the predefined rules.

The rules collectively enable essential ICMPv6 traffic for correct network functioning, including address resolution and router communication in an IPv6 environment. With these rules in place and a multi-network policy denying traffic, applications are not expected to experience connectivity issues.

### 4.4.4. Working with multi-network policy

To manage network traffic isolation and security for pods on secondary networks, you can create, edit, view, and delete multi-network policies. Before you work with multi-network policies, you must enable multi-network policy support for your cluster.

#### 4.4.4.1. Creating a multi-network policy using the CLI

To define granular rules describing ingress or egress network traffic allowed for namespaces in your cluster, you can create a multi-network policy.

**Prerequisites**

- Your cluster uses a network plugin that supports **NetworkPolicy** objects, such as the OVN-Kubernetes network plugin, with **mode: NetworkPolicy** set.
- You installed the OpenShift CLI (**oc**).
- You logged in to the cluster with a user with **cluster-admin** privileges.
- You are working in the namespace that the multi-network policy applies to.

**Procedure**

1. Create a policy rule.
   a. Create a **<policy_name>.yaml** file:

      ```
      $ touch <policy_name>.yaml
      ```

      where:

      **<policy_name>**
      Specifies the multi-network policy file name.

   b. Define a multi-network policy in the created file. The following example denies ingress traffic from all pods in all namespaces. This is a fundamental policy, blocking all cross-pod networking other than cross-pod traffic allowed by the configuration of other Network Policies.

      ```
      apiVersion: k8s.cni.cncf.io/v1beta1
      kind: MultiNetworkPolicy
      endif:: multi[]
      metadata:
        name: deny-by-default
        annotations:
          k8s.v1.cni.cncf.io/policy-for:<namespace_name>/<network_name>
      spec:
        podSelector: {}
        policyTypes:
        - Ingress
        ingress: []
      ```

      where:

      **<network_name>**
      Specifies the name of a network attachment definition.
      The following example configuration allows ingress traffic from all pods in the same namespace:

```
apiVersion: k8s.cni.cncf.io/v1beta1
kind: MultiNetworkPolicy
metadata:
  name: allow-same-namespace
  annotations:
    k8s.v1.cni.cncf.io/policy-for:<namespace_name>/<network_name>
spec:
  podSelector:
  ingress:
  - from:
    - podSelector: {}
# ...
```

where:

**<network_name>**

Specifies the name of a network attachment definition.

The following example allows ingress traffic to one pod from a particular namespace. This policy allows traffic to pods that have the **pod-a** label from pods running in **namespace-y**.

```
apiVersion: k8s.cni.cncf.io/v1beta1
kind: MultiNetworkPolicy
metadata:
  name: allow-traffic-pod
  annotations:
    k8s.v1.cni.cncf.io/policy-for:<namespace_name>/<network_name>
spec:
  podSelector:
   matchLabels:
      pod: pod-a
  policyTypes:
  - Ingress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          kubernetes.io/metadata.name: namespace-y
# ...
```

where:

**<network_name>**

Specifies the name of a network attachment definition.

The following example configuration restricts traffic to a service. This policy when applied ensures every pod with both labels **app=bookstore** and **role=api** can only be accessed by pods with label **app=bookstore**. In this example the application could be a REST API server, marked with labels **app=bookstore** and **role=api**.

This example configuration addresses the following use cases:

- Restricting the traffic to a service to only the other microservices that need to use it.
- Restricting the connections to a database to only permit the application using it.

```
apiVersion: k8s.cni.cncf.io/v1beta1
kind: MultiNetworkPolicy
metadata:
  name: api-allow
  annotations:
    k8s.v1.cni.cncf.io/policy-for:<namespace_name>/<network_name>
spec:
  podSelector:
    matchLabels:
      app: bookstore
      role: api
  ingress:
  - from:
      - podSelector:
          matchLabels:
            app: bookstore
# ...
```

where:

**<network_name>**

Specifies the name of a network attachment definition.

2. To create the multi-network policy object, enter the following command. Successful output lists the name of the policy object and the **created** status.

```
$ oc apply -f <policy_name>.yaml -n <namespace>
```

where:

**<policy_name>**

Specifies the multi-network policy file name.

**<namespace>**

Optional parameter. If you defined the object in a different namespace than the current namespace, the parameter specifices the namespace.

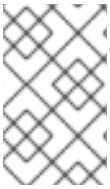
Successful output lists the name of the policy object and the **created** status.

NOTE

If you log in to the web console with **cluster-admin** privileges, you have a choice of creating a network policy in any namespace in the cluster directly in YAML or from a form in the web console.

#### 4.4.4.2. Editing a multi-network policy

To modify existing policy configurations, you can edit a multi-network policy in a namespace. Edit policies by modifying the policy file and applying it with **oc apply**, or by using the **oc edit** command directly.



NOTE

If you log in with **cluster-admin** privileges, you can edit network policies in any namespace in the cluster. In the web console, you can edit policies directly in YAML or by using the **Actions** menu.

**Prerequisites**

- Your cluster uses a network plugin that supports **NetworkPolicy** objects, such as the OVN-Kubernetes network plugin, with **mode: NetworkPolicy** set.
- You installed the OpenShift CLI (**oc**).
- You are logged in to the cluster with a user with **cluster-admin** privileges.
- You are working in the namespace where the multi-network policy exists.

**Procedure**

1. Optional: To list the multi-network policy objects in a namespace, enter the following command:

```
$ oc get multi-network policy -n <namespace>
```

   where:

   **<namespace>**

   Optional: Specifies the namespace if the object is defined in a different namespace than the current namespace.

2. Edit the multi-network policy object.

   a. If you saved the multi-network policy definition in a file, edit the file and make any necessary changes, and then enter the following command.

```
$ oc apply -n <namespace> -f <policy_file>.yaml
```

      where:

      **<namespace>**

      Optional: Specifies the namespace if the object is defined in a different namespace than the current namespace.

      **<policy_file>**

      Specifies the name of the file containing the network policy.

   b. If you need to update the multi-network policy object directly, enter the following command:

```
$ oc edit multi-network policy <policy_name> -n <namespace>
```

      where:

      **<policy_name>**

      Specifies the name of the network policy.

**<namespace>**

Optional: Specifies the namespace if the object is defined in a different namespace than the current namespace.

3. Confirm that the multi-network policy object is updated.

```
$ oc describe multi-networkpolicy <policy_name> -n <namespace>
```

where:

**<policy_name>**

Specifies the name of the multi-network policy.

**<namespace>**

Optional: Specifies the namespace if the object is defined in a different namespace than the current namespace.

#### 4.4.4.3. Viewing multi-network policies using the CLI

You can examine the multi-network policies in a namespace.

NOTE

If you log in with **cluster-admin** privileges, you can edit network policies in any namespace in the cluster. In the web console, you can edit policies directly in YAML or by using the **Actions** menu.

**Prerequisites**

- You installed the OpenShift CLI (**oc**).
- You are logged in to the cluster with a user with **cluster-admin** privileges.
- You are working in the namespace where the multi-network policy exists.

**Procedure**

1. List multi-network policies in a namespace.
   a. To view multi-network policy objects defined in a namespace enter the following command:

```
$ oc get multi-networkpolicy
```

   b. Optional: To examine a specific multi-network policy enter the following command:

```
$ oc describe multi-networkpolicy <policy_name> -n <namespace>
```

   where:

   **<policy_name>**

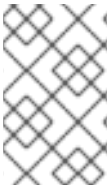
   Specifies the name of the multi-network policy to inspect.

   **<namespace>**

Optional: Specifies the namespace if the object is defined in a different namespace than the current namespace.

#### 4.4.4.4. Deleting a multi-network policy using the CLI

You can delete a multi-network policy in a namespace.



NOTE

If you log in with **cluster-admin** privileges, you can delete network policies in any namespace in the cluster. In the web console, you can delete policies directly in YAML or by using the **Actions** menu.

**Prerequisites**

- Your cluster uses a network plugin that supports **NetworkPolicy** objects, such as the OVN-Kubernetes network plugin, with **mode: NetworkPolicy** set.
- You installed the OpenShift CLI (**oc**).
- You logged in to the cluster with a user with **cluster-admin** privileges.
- You are working in the namespace where the multi-network policy exists.

**Procedure**

- To delete a multi-network policy object, enter the following command. Successful output lists the name of the policy object and the **deleted** status.

```
$ oc delete multi-networkpolicy <policy_name> -n <namespace>
```

where:

**<policy_name>**

Specifies the name of the multi-network policy.

**<namespace>**

Optional parameter. If you defined the object in a different namespace than the current namespace, the parameter specifices the namespace.

#### 4.4.4.5. Creating a default deny all multi-network policy

The default deny all multi-network policy blocks all cross-pod networking other than network traffic allowed by the configuration of other deployed network policies and traffic between host-networked pods. This procedure enforces a strong deny policy by applying a **deny-by-default** policy in the **my-project** namespace.

**WARNING**

Without configuring a **NetworkPolicy** custom resource (CR) that allows traffic communication, the following policy might cause communication problems across your cluster.

**Prerequisites**

- Your cluster uses a network plugin that supports **NetworkPolicy** objects, such as the OVN-Kubernetes network plugin, with **mode: NetworkPolicy** set.
- You installed the OpenShift CLI (**oc**).
- You logged in to the cluster with a user with **cluster-admin** privileges.
- You are working in the namespace that the multi-network policy applies to.

**Procedure**

1. Create the following YAML that defines a **deny-by-default** policy to deny ingress from all pods in all namespaces. Save the YAML in the **deny-by-default.yaml** file:

```
apiVersion: k8s.cni.cncf.io/v1beta1
kind: MultiNetworkPolicy
metadata:
  name: deny-by-default
  namespace: my-project
  annotations:
    k8s.v1.cni.cncf.io/policy-for:<namespace_name>/<network_name>
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  ingress: []
```

where:

**namespace**

Specifies the namespace in which to deploy the policy. For example, the **my-project** namespace.

**annotations**

Specifies the name of namespace project followed by the network attachment definition name.

**podSelector**

If this field is empty, the configuration matches all the pods. Therefore, the policy applies to all pods in the **my-project** namespace.

**policyTypes**

Specifies a list of rule types that the **NetworkPolicy** relates to.

**- Ingress**

Specifies **Ingress** only **policyTypes**.

**ingress**

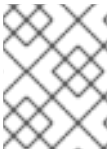
Specifies ingress rules. If not specified, all incoming traffic is dropped to all pods.

2. Apply the policy by entering the following command. Successful output lists the name of the policy object and the **created** status.

```
$ oc apply -f deny-by-default.yaml
```

#### 4.4.4.6. Creating a multi-network policy to allow traffic from external clients

With the **deny-by-default** policy in place you can proceed to configure a policy that allows traffic from external clients to a pod with the label **app=web**.



NOTE

If you log in with a user with the **cluster-admin** role, then you can create a network policy in any namespace in the cluster.

Follow this procedure to configure a policy that allows external service from the public Internet directly or by using a Load Balancer to access the pod. Traffic is only allowed to a pod with the label **app=web**.

**Prerequisites**

- Your cluster uses a network plugin that supports **NetworkPolicy** objects, such as the OVN-Kubernetes network plugin, with **mode: NetworkPolicy** set.
- You installed the OpenShift CLI (**oc**).
- You logged in to the cluster with a user with **cluster-admin** privileges.
- You are working in the namespace that the multi-network policy applies to.

**Procedure**

1. Create a policy that allows traffic from the public Internet directly or by using a load balancer to access the pod. Save the YAML in the **web-allow-external.yaml** file:

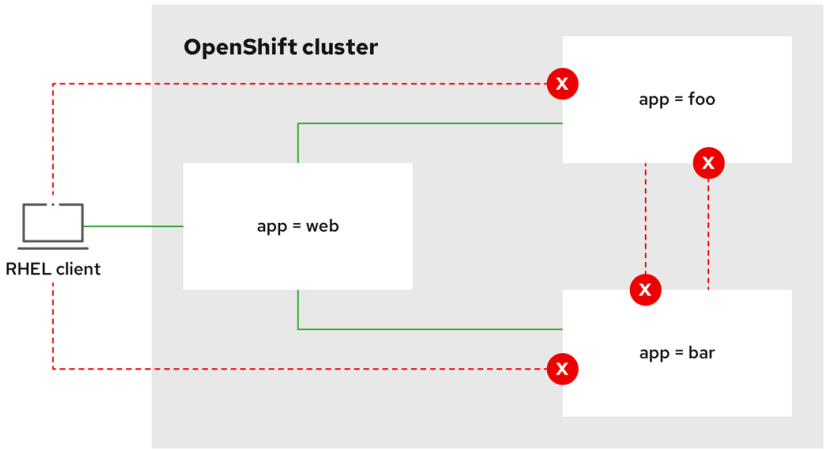
```
apiVersion: k8s.cni.cncf.io/v1beta1
kind: MultiNetworkPolicy
metadata:
  name: web-allow-external
  namespace: default
  annotations:
    k8s.v1.cni.cncf.io/policy-for:<namespace_name>/<network_name>
spec:
  policyTypes:
  - Ingress
  podSelector:
    matchLabels:
```

```
    app: web
ingress:
  - {}
```

2. Apply the policy by entering the following command. Successful output lists the name of the policy object and the **created** status.

```
$ oc apply -f web-allow-external.yaml
```

This policy allows traffic from all resources, including external traffic as illustrated in the following diagram:




### 4.4.4.7. Creating a multi-network policy allowing traffic to an application from all namespaces

You can configure a policy that allows traffic from all pods in all namespaces to a particular application.

NOTE

If you log in with a user with the **cluster-admin** role, then you can create a network policy in any namespace in the cluster.

#### Prerequisites

- Your cluster uses a network plugin that supports **NetworkPolicy** objects, such as the OVN-Kubernetes network plugin, with **mode: NetworkPolicy** set.
- You installed the OpenShift CLI (**oc**).
- You logged in to the cluster with a user with **cluster-admin** privileges.
- You are working in the namespace that the multi-network policy applies to.

#### Procedure

1. Create a policy that allows traffic from all pods in all namespaces to a particular application. Save the YAML in the **web-allow-all-namespaces.yaml** file:

```
apiVersion: k8s.cni.cncf.io/v1beta1
```

```
kind: MultiNetworkPolicy
metadata:
  name: web-allow-all-namespaces
  namespace: default
  annotations:
    k8s.v1.cni.cncf.io/policy-for:<namespace_name>/<network_name>
spec:
  podSelector:
    matchLabels:
      app: web
  policyTypes:
  - Ingress
  ingress:
  - from:
    - namespaceSelector: {}
```

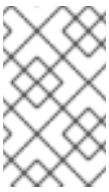
where:

**app**

Applies the policy only to **app:web** pods in default namespace.

**namespaceSelector**

Selects all pods in all namespaces.



NOTE

By default, if you do not specify a **namespaceSelector** parameter in the policy object, no namespaces get selected. This means the policy allows traffic only from the namespace where the network policy deployes.

2. Apply the policy by entering the following command. Successful output lists the name of the policy object and the **created** status.

```
$ oc apply -f web-allow-all-namespaces.yaml
```

### Verification

1. Start a web service in the **default** namespace by entering the following command:

```
$ oc run web --namespace=default --image=nginx --labels="app=web" --expose --port=80
```

2. Run the following command to deploy an **alpine** image in the **secondary** namespace and to start a shell:

```
$ oc run test-$RANDOM --namespace=secondary --rm -i -t --image=alpine -- sh
```

3. Run the following command in the shell and observe that the service allows the request:

```
# wget -qO- --timeout=2 http://web.default
```

```
<!DOCTYPE html>
<html>
```

```
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

#### 4.4.4.8. Creating a multi-network policy allowing traffic to an application from a namespace

You can configure a policy that allows traffic to a pod with the label **app=web** from a particular namespace. This configuration is useful in the following use cases:

- Restrict traffic to a production database only to namespaces that have production workloads deployed.
- Enable monitoring tools deployed to a particular namespace to scrape metrics from the current namespace.

NOTE

If you log in with a user with the **cluster-admin** role, then you can create a network policy in any namespace in the cluster.

**Prerequisites**

- Your cluster uses a network plugin that supports **NetworkPolicy** objects, such as the OVN-Kubernetes network plugin, with **mode: NetworkPolicy** set.
- You installed the OpenShift CLI (**oc**).
- You logged in to the cluster with a user with **cluster-admin** privileges.
- You are working in the namespace that the multi-network policy applies to.

**WARNING**

Do not apply the **network.openshift.io/policy-group: ingress** label to custom namespace or projects. This label is Operator-managed and reserved for OpenShift Container Platform networking functions. It should not be altered on system-created namespaces.

Using this label can result in intermittent network connectivity drops, unintended application of system **NetworkPolicies** resource, or configuration drift as the operator attempts to reconcile the state. For custom traffic grouping, always use unique, user-defined labels as shown in the following procedure.

**Procedure**

1. Create a policy that allows traffic from all pods in a particular namespaces with a label **purpose=production**. Save the YAML in the **web-allow-prod.yaml** file:

```
apiVersion: k8s.cni.cncf.io/v1beta1
kind: MultiNetworkPolicy
metadata:
  name: web-allow-prod
  namespace: default
  annotations:
    k8s.v1.cni.cncf.io/policy-for:<namespace_name>/<network_name>
spec:
  podSelector:
    matchLabels:
      app: web
  policyTypes:
  - Ingress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          purpose: production
```

where:

**app**

Applies the policy only to **app:web** pods in the default namespace.

**purpose**

Restricts traffic to only pods in namespaces that have the label **purpose=production**.

2. Apply the policy by entering the following command. Successful output lists the name of the policy object and the **created** status.

```
$ oc apply -f web-allow-prod.yaml
```

**Verification**

1. Start a web service in the **default** namespace by entering the following command:

```
$ oc run web --namespace=default --image=nginx --labels="app=web" --expose --port=80
```

2. Run the following command to create the **prod** namespace:

```
$ oc create namespace prod
```

3. Run the following command to label the **prod** namespace:

```
$ oc label namespace/prod purpose=production
```

4. Run the following command to create the **dev** namespace:

```
$ oc create namespace dev
```

5. Run the following command to label the **dev** namespace:

```
$ oc label namespace/dev purpose=testing
```

6. Run the following command to deploy an **alpine** image in the **dev** namespace and to start a shell:

```
$ oc run test-$RANDOM --namespace=dev --rm -i -t --image=alpine -- sh
```

7. Run the following command in the shell and observe the reason for the blocked request. For example, expected output states **wget: download timed out**.

```
# wget -qO- --timeout=2 http://web.default
```

8. Run the following command to deploy an **alpine** image in the **prod** namespace and start a shell:

```
$ oc run test-$RANDOM --namespace=prod --rm -i -t --image=alpine -- sh
```

9. Run the following command in the shell and observe that the request is allowed:

```
# wget -qO- --timeout=2 http://web.default
```

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>
```

```
<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

### 4.4.5. Additional resources

- About network policy
- Understanding multiple networks
- Configuring a macvlan network
- Configuring an SR-IOV network device

## 4.5. REMOVING A POD FROM A SECONDARY NETWORK

To disconnect a pod from specific network configurations in OpenShift Container Platform, you can remove the pod from a secondary network. Delete the pod to remove its connection to the secondary network.

### 4.5.1. Removing a pod from a secondary network

To disconnect a pod from specific network configurations in OpenShift Container Platform, you can remove the pod from a secondary network. Delete the pod using the **oc delete pod** command to remove its connection to the secondary network.

#### Prerequisites

- A secondary network is attached to the pod.
- Install the OpenShift CLI (**oc**).
- Log in to the cluster.

#### Procedure

- Delete the pod by entering the following command:

```
$ oc delete pod <name> -n <namespace>
```

where:

**<name>**
: Specifies the name of the pod.

**<namespace>**
: Specifies the namespace that contains the pod.

# 4.6. EDITING A SECONDARY NETWORK

To update network settings or change network parameters for a secondary network in OpenShift Container Platform, you can modify the configuration for an existing secondary network. Edit the **NetworkAttachmentDefinition** custom resource to apply your changes.

## 4.6.1. Modifying a NetworkAttachmentDefinition custom resource

To update network settings or change network parameters for a secondary network in OpenShift Container Platform, you can modify the **NetworkAttachmentDefinition** custom resource. Edit the Cluster Network Operator CR to apply your changes.

### Prerequisites

- You have configured a secondary network for your cluster.
- Install the OpenShift CLI (**oc**).
- Log in as a user with **cluster-admin** privileges.

### Procedure

1. Edit the Cluster Network Operator (CNO) CR in your default text editor by running the following command:

```
$ oc edit networks.operator.openshift.io cluster
```

2. In the **additionalNetworks** collection, update the secondary network with your changes.
3. Save your changes and quit the text editor to commit your changes.
4. Optional: Confirm that the CNO updated the **NetworkAttachmentDefinition** object by running the following command. Replace **<network_name>** with the name of the secondary network to display. There might be a delay before the CNO updates the **NetworkAttachmentDefinition** object to reflect your changes.

```
$ oc get network-attachment-definitions <network_name> -o yaml
```

For example, the following console output displays a **NetworkAttachmentDefinition** object that is named **net1**:

```
$ oc get network-attachment-definitions net1 -o go-template='{{printf "%s\n" .spec.config}}'
{ "cniVersion": "0.3.1", "type": "macvlan",
"master": "ens5",
"mode": "bridge",
"ipam":       {"type":"static","routes":[{"dst":"0.0.0.0/0","gw":"10.128.2.1"}],"addresses":
[{"address":"10.128.2.100/23","gateway":"10.128.2.1"}],"dns":{"nameservers":
["172.30.0.10"],"domain":"us-west-2.compute.internal","search":["us-west-
2.compute.internal"]}} }
```

## 4.6.2. Using an OVN-Kubernetes localnet topology to map VLANs to a secondary interface

You can use OVN-Kubernetes localnet topology in a **NetworkAttachmentDefinition** (NAD) to map a specific VLAN ID from the physical network to the secondary interface of a pod.

To provide multiple VLANs for cluster workloads in OpenShift Container Platform, define additional VLANs in the **NetworkAttachmentDefinition** custom resource (CR). Configuring trunk ports ensures that the physical network associates correctly with your virtual infrastructure for reliable traffic management.

The example in the procedure demonstrates the following configurations:

- Physical switch ports connect to OpenShift Container Platform nodes by using VLAN trunking. The trunk carries tagged traffic for the VLANs you define in NADs.
- The **br-ex** acts as the OVS bridge that connects virtual workloads to the physical workloads.
- Multiple NADs with specific VLAN tags get created by using the **localnet** topology. This configuration defines specific VLAN IDs for traffic isolation.
- Pods or virtual machines (VMs) attach to the NAD CRs for the purposes of improved network connectivity.

### Prerequisites

- You installed the OpenShift CLI (**oc**).
- You logged in as a user with **cluster-admin** privileges.
- You installed the NMState Operator.
- You configured the **br-ex** bridge interface during cluster installation.

### Procedure

1. Create an **NetworkAttachmentDefinition** CR for each VLAN, such as **nad-cvlan100.yaml**. OVN-Kubernetes uses the NAD files to tag and untag Ethernet frames for pods or VMs.

#### Example configuration

```
apiVersion: k8s.cni.cncf.io/v1
kind: NetworkAttachmentDefinition
metadata:
  name: vlan-100
  namespace: default
spec:
  config: |-
    {
      "cniVersion": "0.4.0",
      "name": "localnet-vlan-100",
      "type": "ovn-k8s-cni-overlay",
      "physicalNetworkName": "physnet",
      "topology": "localnet",
      "vlanID": 100,
      "mtu": 1500,
      "netAttachDefName": "default/vlan-100"
    }
# ...
```

2. Attach pods or VMs to the VLANs by referencing the NAD in the configuration for the pod or VM:

**Example pod configuration**

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    k8s.v1.cni.cncf.io/networks: vlan-100
# ...
```

**Example VM configuration**

```
apiVersion: kubevirt.io/v1
kind: VirtualMachine
spec:
    template:
        spec:
        networks:
        - multus:
            networkName: vlan-100
            name: secondary-vlan
# ...
```

## 4.7. CONFIGURING IP ADDRESS ASSIGNMENT ON SECONDARY NETWORKS

You can configure IP address assignments for secondary networks so that pods can connect to the secondary networks.

### 4.7.1. Configuration of IP address assignment for a network attachment

For secondary networks, you can assign IP addresses by using an IP Address Management (IPAM) CNI plugin, which supports various assignment methods, including Dynamic Host Configuration Protocol (DHCP) and static assignment.

The DHCP IPAM CNI plugin responsible for dynamic assignment of IP addresses operates with two distinct components:

- CNI Plugin: Responsible for integrating with the Kubernetes networking stack to request and release IP addresses.
- DHCP IPAM CNI Daemon: A listener for DHCP events that coordinates with existing DHCP servers in the environment to handle IP address assignment requests. This daemon is not a DHCP server itself.

For networks requiring **type: dhcp** in their IPAM configuration, ensure the DHCP server meets the following conditions:

- A DHCP server is available and running in the environment.

- The DHCP server is external to the cluster and you expect the server to form part of the existing network infrastructure for the customer.
- The DHCP server is appropriately configured to serve IP addresses to the nodes.

In cases where a DHCP server is unavailable in the environment, consider using the Whereabouts IPAM CNI plugin. The Whereabouts CNI provides similar IP address management capabilities without the need for an external DHCP server.

NOTE

Use the Whereabouts CNI plugin when no external DHCP server exists or where static IP address management is preferred. The Whereabouts plugin includes a reconciler daemon to manage stale IP address allocations.

Ensure the periodic renewal of a DHCP lease throughout the lifetime of a container by including a separate daemon, the DHCP IPAM CNI Daemon. To deploy the DHCP IPAM CNI daemon, change the Cluster Network Operator (CNO) configuration to trigger the deployment of this daemon as part of the secondary network setup.

#### 4.7.1.1. Static IP address assignment configuration

The following table describes the configuration for static IP address assignment:

**Table 4.3. ipam static configuration object**

| Field | Type | Description |
|---|---|---|
| **type** | **string** | The IPAM address type. The value **static** is required. |
| **addresses** | **array** | An array of objects specifying IP addresses to assign to the virtual interface. Both IPv4 and IPv6 IP addresses are supported. |
| **routes** | **array** | An array of objects specifying routes to configure inside the pod. |
| **dns** | **array** | Optional: An array of objects specifying the DNS configuration. |

The **addresses** array requires objects with the following fields:

**Table 4.4. ipam.addresses[] array**

| Field | Type | Description |
|---|---|---|
| **address** | **string** | An IP address and network prefix that you specify. For example, if you specify **10.10.21.10/24**, the secondary network gets assigned an IP address of **10.10.21.10** and the netmask of **255.255.255.0**. |
| **gateway** | **string** | The default gateway to route egress network traffic to. |

Table 4.5. **ipam.routes[]** array

| Field | Type | Description |
|---|---|---|
| **dst** | **string** | The IP address range in CIDR format, such as **192.168.17.0/24** or **0.0.0.0/0** for the default route. |
| **gw** | **string** | The gateway that routes network traffic. |

Table 4.6. **ipam.dns** object

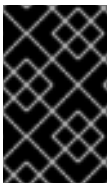
| Field | Type | Description |
|---|---|---|
| **nameservers** | **array** | An array of one or more IP addresses where DNS queries get sent. |
| **domain** | **array** | The default domain to append to a hostname. For example, if the domain is set to **example.com**, a DNS lookup query for **example-host** is rewritten as **example-host.example.com**. |
| **search** | **array** | An array of domain names to append to an unqualified hostname, such as **example-host**, during a DNS lookup query. |

### Static IP address assignment configuration example

```
{
  "ipam": {
    "type": "static",
      "addresses": [
        {
          "address": "191.168.1.7/24"
        }
      ]
  }
}
```

### 4.7.1.2. Dynamic IP address (DHCP) assignment configuration

A pod obtains its original DHCP lease when the pod gets created. The lease must be periodically renewed by a minimal DHCP server deployment running on the cluster.



**IMPORTANT**

For an Ethernet network attachment, the SR-IOV Network Operator does not create a DHCP server deployment; the Cluster Network Operator is responsible for creating the minimal DHCP server deployment.

To trigger the deployment of the DHCP server, you must create a shim network attachment by editing the Cluster Network Operator configuration, as in the following example:

#### Example shim network attachment definition

```
apiVersion: operator.openshift.io/v1
kind: Network
metadata:
  name: cluster
spec:
  additionalNetworks:
  - name: dhcp-shim
    namespace: default
    type: Raw
    rawCNIConfig: |-
      {
        "name": "dhcp-shim",
        "cniVersion": "0.3.1",
        "type": "bridge",
        "ipam": {
          "type": "dhcp"
        }
      }
  # ...
```

where:

**type**

Specifies dynamic IP address assignment for the cluster.

### 4.7.1.3. Dynamic IP address assignment configuration with Whereabouts

The Whereabouts CNI plugin helps the dynamic assignment of an IP address to a secondary network without the use of a DHCP server.

The Whereabouts CNI plugin also supports overlapping IP address ranges and configuration of the same CIDR range multiple times within separate **NetworkAttachmentDefinition** CRDs. This provides greater flexibility and management capabilities in multitenant environments.

#### 4.7.1.3.1. Dynamic IP address configuration parameters

The following table describes the configuration objects for dynamic IP address assignment with Whereabouts:

**Table 4.7. ipam whereabouts configuration parameters**

| Field | Type | Description |
|---|---|---|
| **type** | **string** | The IPAM address type. The value **whereabouts** is required. |
| **range** | **string** | An IP address and range in CIDR notation. IP addresses are assigned from within this range of addresses. |
| **exclude** | **array** | Optional: A list of zero or more IP addresses and ranges in CIDR notation. IP addresses within an excluded address range are not assigned. |

| Field | Type | Description |
| --- | --- | --- |
| **network_name** | **string** | Optional: Helps ensure that each group or domain of pods gets its own set of IP addresses, even if they share the same range of IP addresses. Setting this field is important for keeping networks separate and organized, notably in multi-tenant environments. |

#### 4.7.1.3.2. Dynamic IP address assignment configuration with Whereabouts that excludes IP address ranges

The following example shows a dynamic address assignment configuration in a NAD file that uses Whereabouts:

**Whereabouts dynamic IP address assignment that excludes specific IP address ranges**

```
{
  "ipam": {
    "type": "whereabouts",
    "range": "192.0.2.192/27",
    "exclude": [
      "192.0.2.192/30",
      "192.0.2.196/32"
    ]
  }
}
```

#### 4.7.1.3.3. Dynamic IP address assignment that uses Whereabouts with overlapping IP address ranges

The following example shows a dynamic IP address assignment that uses overlapping IP address ranges for multitenant networks.

**NetworkAttachmentDefinition 1**

```
{
  "ipam": {
    "type": "whereabouts",
    "range": "192.0.2.192/29",
    "network_name": "example_net_common",
  }
}
```

where:

**network_name**

Optional parameter. If set, must match the **network_name** of **NetworkAttachmentDefinition 2**.

**NetworkAttachmentDefinition 2**

```
{
```

```
  "ipam": {
    "type": "whereabouts",
    "range": "192.0.2.192/24",
    "network_name": "example_net_common",
  }
}
```

where:

**network_name**

Optional parameter. If set, must match the **network_name** of **NetworkAttachmentDefinition 1**.

#### 4.7.1.4. Creating a whereabouts-reconciler daemon set

The Whereabouts reconciler is responsible for managing dynamic IP address assignments for the pods within a cluster by using the Whereabouts IP Address Management (IPAM) solution. The Whereabouts reconciler ensures that each pod gets a unique IP address from the specified IP address range. The Whereabouts reconciler also handles IP address releases when pods are deleted or scaled down.

NOTE

You can also use a **NetworkAttachmentDefinition** custom resource definition (CRD) for dynamic IP address assignment.

The **whereabouts-reconciler** daemon set is automatically created when you configure a secondary network through the Cluster Network Operator. The **whereabouts-reconciler** DaemonSet does not get automatically created when you configure a secondary network from a YAML manifest.

To trigger the deployment of the **whereabouts-reconciler** daemon set, you must manually create a **whereabouts-shim** network attachment by editing the Cluster Network Operator custom resource (CR) file.

**Procedure**

1. Edit the **Network.operator.openshift.io** CR by running the following command:

```
$ oc edit network.operator.openshift.io cluster
```

2. Include the **additionalNetworks** section shown in this example YAML extract within the **spec** definition of the CR:

```
apiVersion: operator.openshift.io/v1
kind: Network
metadata:
  name: cluster
# ...
spec:
  additionalNetworks:
  - name: whereabouts-shim
    namespace: default
    rawCNIConfig: |-
      {
        "name": "whereabouts-shim",
```

```
      "cniVersion": "0.3.1",
      "type": "bridge",
      "ipam": {
        "type": "whereabouts"
      }
    }
  type: Raw
# ...
```

3. Save the file and exit the text editor.

4. Verify that the **whereabouts-reconciler** daemon set deployed successfully by running the following command:

```
$ oc get all -n openshift-multus | grep whereabouts-reconciler
```

```
pod/whereabouts-reconciler-jnp6g 1/1 Running 0 6s
pod/whereabouts-reconciler-k76gg 1/1 Running 0 6s
daemonset.apps/whereabouts-reconciler 6 6 6 6 6 kubernetes.io/os=linux 6s
```

#### 4.7.1.5. Configuring the Whereabouts IP reconciler schedule

The Whereabouts IPAM CNI plugin runs the IP address reconciler daily. This process cleans up any stranded IP address allocations that might result in exhausting IP addresses and therefore prevent new pods from getting a stranded IP address allocated to them.

Use this procedure to change the frequency at which the IP reconciler runs.

**Prerequisites**

- You installed the OpenShift CLI (**oc**).
- You have access to the cluster as a user with the **cluster-admin** role.
- You have deployed the **whereabouts-reconciler** daemon set, and the **whereabouts-reconciler** pods are up and running.

**Procedure**

1. Run the following command to create a **ConfigMap** object named **whereabouts-config** in the **openshift-multus** namespace with a specific cron expression for the IP reconciler:

```
$ oc create configmap whereabouts-config -n openshift-multus --from-
literal=reconciler_cron_expression="*/15 * * * *"
```

This cron expression indicates the IP reconciler runs every 15 minutes. Adjust the expression based on your specific requirements.

NOTE

The **whereabouts-reconciler** daemon set can only consume a cron expression pattern that includes five asterisks. Red Hat does not support the sixth asterisk, which is used to denote seconds.

2. Retrieve information about resources related to the **whereabouts-reconciler** daemon set and pods within the **openshift-multus** namespace by running the following command:

```
$ oc get all -n openshift-multus | grep whereabouts-reconciler
```

```
pod/whereabouts-reconciler-2p7hw                 1/1     Running   0          4m14s
pod/whereabouts-reconciler-76jk7                 1/1     Running   0          4m14s
daemonset.apps/whereabouts-reconciler        6         6         6       6            6
kubernetes.io/os=linux   4m16s
```

3. Run the following command to verify that the **whereabouts-reconciler** pod runs the IP reconciler with the configured interval:

```
$ oc -n openshift-multus logs whereabouts-reconciler-2p7hw
```

```
2024-02-02T16:33:54Z [debug] event not relevant: "/cron-
schedule/..2024_02_02_16_33_54.1375928161": CREATE
2024-02-02T16:33:54Z [debug] event not relevant: "/cron-
schedule/..2024_02_02_16_33_54.1375928161": CHMOD
2024-02-02T16:33:54Z [debug] event not relevant: "/cron-schedule/..data_tmp": RENAME
2024-02-02T16:33:54Z [verbose] using expression: */15 * * * *
2024-02-02T16:33:54Z [verbose] configuration updated to file "/cron-schedule/..data". New
cron expression: */15 * * * *
2024-02-02T16:33:54Z [verbose] successfully updated CRON configuration id "00c2d1c9-
631d-403f-bb86-73ad104a6817" - new cron expression: */15 * * * *
2024-02-02T16:33:54Z [debug] event not relevant: "/cron-schedule/config": CREATE
2024-02-02T16:33:54Z [debug] event not relevant: "/cron-
schedule/..2024_02_02_16_26_17.3874177937": REMOVE
2024-02-02T16:45:00Z [verbose] starting reconciler run
2024-02-02T16:45:00Z [debug] NewReconcileLooper - inferred connection data
2024-02-02T16:45:00Z [debug] listing IP pools
2024-02-02T16:45:00Z [debug] no IP addresses to cleanup
2024-02-02T16:45:00Z [verbose] reconciler success
```

#### 4.7.1.6. Fast IPAM configuration for the Whereabouts IPAM CNI plugin

Wherabouts is an IP Address Management (IPAM) Container Network Interface (CNI) plugin that assigns IP addresses at a cluster-wide level. Whereabouts does not require a Dynamic Host Configuration Protocol (DHCP) server.

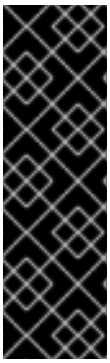
A typical Wherabouts workflow is described as follows:

1. Whereabouts takes an address range in classless inter-domain routing (CIDR) notation, such as **192.168.2.0/24**, and assigns IP addresses within that range, such as **192.168.2.1** to **192.168.2.254**.

2. Whereabouts assigns an IP address, the lowest value address in a CIDR range, to a pod and tracks the IP address in a data store for the lifetime of that pod.

3. When the pod is removed, Whereabouts frees the address from the pod so that the address is available for assignment.

To improve the performance of Whereabouts, especially if nodes in your cluster run a high amount of pods, you can enable the Fast IPAM feature.



**IMPORTANT**

Fast IPAM is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

The Fast IPAM feature uses **nodeslicepools**, which are managed by the Whereabouts Controller, to optimize IP allocation for nodes.

### Prerequisites

- You added the **whereabouts-shim** configuration to the **Network.operator.openshift.io** custom resource (CR), so that the Cluster Network Operator (CNO) can deploy the Whereabouts Controller. See "Creating a Whereabouts reconciler daemon set".
- For the Fast IPAM feature to work, ensure that the **NetworkAttachmentDefinition** (NAD) and the pod exist in the same **openshift-multus** namespace.

### Procedure

1. Confirm that the Whereabouts Controller is running by entering the following command.

```
$ oc get pods -n openshift-multus | grep whereabouts-controller
```

```
whereabouts-controller-5cbfd6c475-fr7d7       1/1     Running            0               22s
...
```

**IMPORTANT**

If the Whereabouts Controller is not running, the Fast IPAM does not work.

2. Create a NAD file for your cluster and add the Fast IPAM details to the file as demonstrated in the following example configuration:

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: wb-ipam
  namespace: openshift-multus
spec:
  config: '{
 "cniVersion": "0.3.0",
 "name": "wb-ipam-cni-name",
 "type": "bridge",
 "bridge": "cni0",
 "ipam": {
   "type": "whereabouts",
   "range": "10.5.0.0/20",
```

```
    "node_slice_size": "/24"
     }
   }'
  # ...
```

where:

**namespace**

The namespace where CNO deploys the NAD.

**name**

The name of the Whereabouts IPAM CNI plugin.

**type**

The type of IPAM CNI plugin, such as **whereabouts**.

**range**

The IP address range for the IP pool that the Whereabouts IPAM CNI plugin uses for allocating IP addresses to pods.

**node_slice_size**

Sets the slice size of IP addresses available to each node.

3. Add the Whereabouts IPAM CNI plugin annotation details to the YAML file for the pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: samplepod
  annotations:
  k8s.v1.cni.cncf.io/networks: openshift-multus/wb-ipam
spec:
  containers:
  - name: samplecontainer
  command: ["/bin/bash", "-c", "trap : TERM INT; sleep infinity & wait"]
  image: registry.redhat.io/ubi9/ubi-minimal
# ...
```

where:

**name**

The name of the pod.

**k8s.v1.cni.cncf.io/networks**

The annotation details that references the Whereabouts IPAM CNI plugin name that exists in the **openshift-multus** namespace.

**- name**

The name of the container for the pod.

**command**

Defines the entry point for the container and controls the behavior of the container in the Whereabouts IPAM CNI plugin.

4. Apply the NAD file configuration to pods that exist on nodes that run in your cluster:

```
$ oc create -f <NAD_file_name>.yaml
```

- 

### Verification

1. Show the IP address details of the pod by entering the following command:

```
$ oc describe pod <pod_name>
```

```
...
k8s.v1.cni.cncf.io/network-status:
  [{
      "name": "ovn-kubernetes",
      "interface": "eth0",
      "ips": [
          "10.128.3.174"
      ],
      "mac": "0a:58:0a:80:03:ae",
      "default": true,
      "dns": {}
  },{
      "name": "openshift-multus/wb-ipam",
      "interface": "net1",
      "ips": [
          "10.5.0.1"
      ],
      "mac": "1a:04:6f:a4:15:3c",
      "dns": {}
  }]
k8s.v1.cni.cncf.io/networks: openshift-multus/wb-ipam
...
```

2. Access the pod and confirm its interfaces by entering the following command:

```
$ oc exec <pod_name> -- ip a
```

```
...
3: net1@if439: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default qlen 1000
    link/ether 1a:04:6f:a4:15:3c brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.5.0.1/20 brd 10.5.15.255 scope global net1
       valid_lft forever preferred_lft forever
    inet6 fe80::1804:6fff:fea4:153c/64 scope link
       valid_lft forever preferred_lft forever
...
```

where:

**inet**: Pod is attached to the **10.5.0.1** IP address on the **net1** interface as expected.

3. Check that the node selector pool exists in the **openshift-multus** namespace by entering the following command. The expected output shows the name of the node selector pool, such as **nodeslicepool, and the creation age in minutes, such as `32m**.

```
$ oc get nodeslicepool -n openshift-multus
```

**Example output**

```
NAME                AGE
wb-ipam-cni-name    32m
```

#### 4.7.1.7. Creating a configuration for assignment of dual-stack IP addresses dynamically

You can dynamically assign dual-stack IP addresses to a secondary network so that pods can communicate over both IPv4 and IPv6 addresses.

You can configure the following IP address assignment types in the **ipRanges** parameter:

- IPv4 addresses
- IPv6 addresses
- multiple IP address assignment

**Procedure**

1. Set **type** to **whereabouts**.
2. Use **ipRanges** to allocate IP addresses as shown in the following example:

```
cniVersion: operator.openshift.io/v1
kind: Network
metadata:
  name: cluster
spec:
  additionalNetworks:
  - name: whereabouts-shim
    namespace: default
    type: Raw
    rawCNIConfig: |-
      {
       "name": "whereabouts-dual-stack",
       "cniVersion": "0.3.1,
       "type": "bridge",
       "ipam": {
         "type": "whereabouts",
         "ipRanges": [
                {"range": "192.168.10.0/24"},
                {"range": "2001:db8::/64"}
            ]
       }
      }
```

3. Attach the secondary network to a pod. For more information, see "Adding a pod to a secondary network".

**Verification**

- Verify that all IP addresses got assigned to the network interfaces within the network namespace of a pod by entering the following command:

```
$ oc exec -it <pod_name> -- ip a
```

where:

**<podname>**

The name of the pod.

## 4.8. CONFIGURING THE MASTER INTERFACE IN THE CONTAINER NETWORK NAMESPACE

You can create and manage a MAC-VLAN, IP-VLAN, and VLAN subinterface based on a **master** interface.

### 4.8.1. About configuring the master interface in the container network namespace

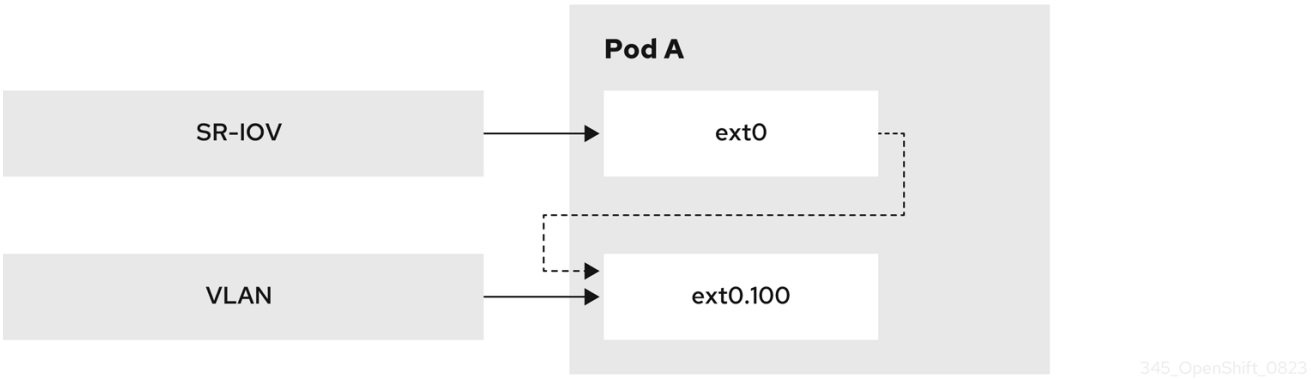
You can create a MAC-VLAN, an IP-VLAN, or a VLAN subinterface that is based on a **master** interface that exists in a container namespace. You can also create a **master** interface as part of the pod network configuration in a separate network attachment definition CRD.

To use a container namespace **master** interface, you must specify **true** for the **linkInContainer** parameter that exists in the subinterface configuration of the **NetworkAttachmentDefinition** CRD.

#### 4.8.1.1. Creating multiple VLANs on SR-IOV VFs

You can create multiple VLANs based on SR-IOV VFs. For this configuration, create an SR-IOV network and then define the network attachments for the VLAN interfaces.

The following diagram shows the setup process for creating multiple VLANs on SR-IOV VFs.




**Prerequisites**

- You installed the OpenShift CLI (**oc**).
- You have access to the cluster as a user with the **cluster-admin** role.
- You have installed the SR-IOV Network Operator.

**Procedure**

1. Create a dedicated container namespace where you want to deploy your pod by using the following command:

```
$ oc new-project test-namespace
```

2. Create an SR-IOV node policy.

   a. Create an **SriovNetworkNodePolicy** object, and then save the YAML in the **sriov-node-network-policy.yaml** file:

```
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetworkNodePolicy
metadata:
 name: sriovnic
 namespace: openshift-sriov-network-operator
spec:
 deviceType: netdevice
 isRdma: false
 needVhostNet: true
 nicSelector:
   vendor: "15b3"
   deviceID: "101b"
   rootDevices: ["00:05.0"]
 numVfs: 10
 priority: 99
 resourceName: sriovnic
 nodeSelector:
    feature.node.kubernetes.io/network-sriov.capable: "true"
```

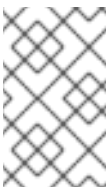
   where:

   **vendor**

   The vendor hexadecimal code of the SR-IOV network device. The value **15b3** associates with a Mellanox NIC.

   **deviceID**

   The device hexadecimal code of the SR-IOV network device.

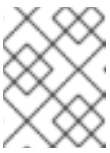


NOTE

The SR-IOV network node policy configuration example, with the setting **deviceType: netdevice**, is tailored specifically for Mellanox Network Interface Cards (NICs).

   b. Apply the YAML configuration by running the following command:

```
$ oc apply -f sriov-node-network-policy.yaml
```



NOTE

Applying the YAML configuration might take time because of a node reboot operation.

3. Create an SR-IOV network:

   a. Create the **SriovNetwork** custom resource (CR) for the additional secondary SR-IOV network attachment as demonstrated in the following example CR. Save the YAML as a **sriov-network-attachment.yaml** file:

```
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetwork
metadata:
 name: sriov-network
 namespace: openshift-sriov-network-operator
spec:
 networkNamespace: test-namespace
 resourceName: sriovnic
 spoofChk: "off"
 trust: "on"
```

   b. Apply the YAML by running the following command:

```
$ oc apply -f sriov-network-attachment.yaml
```

4. Create the VLAN secondary network.

   a. Using the following YAML example, create a file named **vlan100-additional-network-configuration.yaml**:

```
apiVersion: k8s.cni.cncf.io/v1
kind: NetworkAttachmentDefinition
metadata:
  name: vlan-100
  namespace: test-namespace
spec:
  config: |
    {
      "cniVersion": "0.4.0",
      "name": "vlan-100",
      "plugins": [
        {
          "type": "vlan",
          "master": "ext0",
          "mtu": 1500,
          "vlanId": 100,
          "linkInContainer": true,
          "ipam": {"type": "whereabouts", "ipRanges": [{"range": "1.1.1.0/24"}]}
        }
      ]
    }
```

   where:

   **master**

   The VLAN configuration needs to specify the **master** name. You can specify the name in the networks annotation of a pod.

   **linkInContainer**

The **linkInContainer** parameter must be specified.

b. Apply the YAML file by running the following command:

```
$ oc apply -f vlan100-additional-network-configuration.yaml
```

5. Create a pod definition by using the earlier specified networks.

a. Using the following YAML configuration example, create a file named **pod-a.yaml** file:

NOTE

The manifest example includes the following resources:

- Namespace with security labels
- Pod definition with appropriate network annotation

```
apiVersion: v1
kind: Namespace
metadata:
  name: test-namespace
  labels:
    pod-security.kubernetes.io/enforce: privileged
    pod-security.kubernetes.io/audit: privileged
    pod-security.kubernetes.io/warn: privileged
    security.openshift.io/scc.podSecurityLabelSync: "false"
---
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  namespace: test-namespace
  annotations:
    k8s.v1.cni.cncf.io/networks: '[
      {
        "name": "sriov-network",
        "namespace": "test-namespace",
        "interface": "ext0"
      },
      {
        "name": "vlan-100",
        "namespace": "test-namespace",
        "interface": "ext0.100"
      }
    ]'
spec:
  securityContext:
    runAsNonRoot: true
  containers:
    - name: nginx-container
      image: nginxinc/nginx-unprivileged:latest
      securityContext:
        allowPrivilegeEscalation: false
        capabilities:
```

```
      drop: ["ALL"]
  ports:
    - containerPort: 80
  seccompProfile:
    type: "RuntimeDefault"
```

where:

**interface**

The name to be used as the **master** interface for the VLAN interface.

b. Apply the YAML file by running the following command:

```
$ oc apply -f pod-a.yaml
```

6. Get detailed information about the **nginx-pod** within the **test-namespace** by running the following command:

```
$ oc describe pods nginx-pod -n test-namespace
```

```
Name:         nginx-pod
Namespace:    test-namespace
Priority:     0
Node:         worker-1/10.46.186.105
Start Time:   Mon, 14 Aug 2023 16:23:13 -0400
Labels:       <none>
Annotations:  k8s.ovn.org/pod-networks:
                {"default":{"ip_addresses":
["10.131.0.26/23"],"mac_address":"0a:58:0a:83:00:1a","gateway_ips":["10.131.0.1"],"routes":
[{"dest":"10.128.0.0...
              k8s.v1.cni.cncf.io/network-status:
                [{
                    "name": "ovn-kubernetes",
                    "interface": "eth0",
                    "ips": [
                        "10.131.0.26"
                    ],
                    "mac": "0a:58:0a:83:00:1a",
                    "default": true,
                    "dns": {}
                },{
                    "name": "test-namespace/sriov-network",
                    "interface": "ext0",
                    "mac": "6e:a7:5e:3f:49:1b",
                    "dns": {},
                    "device-info": {
                        "type": "pci",
                        "version": "1.0.0",
                        "pci": {
                            "pci-address": "0000:d8:00.2"
                        }
                    }
                },{
                    "name": "test-namespace/vlan-100",
                    "interface": "ext0.100",
```

```
                "ips": [
                    "1.1.1.1"
                ],
                "mac": "6e:a7:5e:3f:49:1b",
                "dns": {}
            }]
          k8s.v1.cni.cncf.io/networks:
            [ { "name": "sriov-network", "namespace": "test-namespace", "interface": "ext0" }, {
"name": "vlan-100", "namespace": "test-namespace", "i...
          openshift.io/scc: privileged
Status:       Running
IP:           10.131.0.26
IPs:
  IP:  10.131.0.26
```

#### 4.8.1.2. Creating a subinterface based on a bridge master interface in a container namespace

You can create a subinterface based on a bridge **master** interface that exists in a container namespace. Creating a subinterface can be applied to other types of interfaces.

**Prerequisites**

- You have installed the OpenShift CLI (**oc**).
- You are logged in to the OpenShift Container Platform cluster as a user with **cluster-admin** privileges.

**Procedure**

1. Create a dedicated container namespace where you want to deploy your pod by entering the following command:

```
$ oc new-project test-namespace
```

2. Using the following YAML configuration example, create a bridge **NetworkAttachmentDefinition** custom resource definition (CRD) file named **bridge-nad.yaml**:

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: bridge-network
spec:
  config: '{
    "cniVersion": "0.4.0",
    "name": "bridge-network",
    "type": "bridge",
    "bridge": "br-001",
    "isGateway": true,
    "ipMasq": true,
    "hairpinMode": true,
    "ipam": {
      "type": "host-local",
      "subnet": "10.0.0.0/24",
```

```
    "routes": [{"dst": "0.0.0.0/0"}]
  }
}'
```

3. Run the following command to apply the **NetworkAttachmentDefinition** CRD to your OpenShift Container Platform cluster:

```
$ oc apply -f bridge-nad.yaml
```

4. Verify that you successfully created a **NetworkAttachmentDefinition** CRD by entering the following command. The expected output shows the name of the NAD CRD and the creation age in minutes.

```
$ oc get network-attachment-definitions
```

5. Using the following YAML example, create a file named **ipvlan-additional-network-configuration.yaml** for the IPVLAN secondary network configuration:

```
apiVersion: k8s.cni.cncf.io/v1
kind: NetworkAttachmentDefinition
metadata:
  name: ipvlan-net
  namespace: test-namespace
spec:
  config: '{
    "cniVersion": "0.3.1",
    "name": "ipvlan-net",
    "type": "ipvlan",
    "master": "net1",
    "mode": "l3",
    "linkInContainer": true,
    "ipam": {"type": "whereabouts", "ipRanges": [{"range": "10.0.0.0/24"}]}
  }'
```

where:

**master**

Specifies the ethernet interface to associate with the network attachment. The ethernet interface is subsequently configured in the pod networks annotation.

**linkInContainer**

Specifies that the **master** interface exists in the container network namespace.

6. Apply the YAML file by running the following command:

```
$ oc apply -f ipvlan-additional-network-configuration.yaml
```

7. Verify that the **NetworkAttachmentDefinition** CRD has been created successfully by running the following command. The expected output shows the name of the NAD CRD and the creation age in minutes.

```
$ oc get network-attachment-definitions
```

8. Using the following YAML configuration example, create a file named **pod-a.yaml** for the pod definition:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-a
  namespace: test-namespace
  annotations:
    k8s.v1.cni.cncf.io/networks: '[
      {
        "name": "bridge-network",
        "interface": "net1" 1
      },
      {
        "name": "ipvlan-net",
        "interface": "net2"
      }
    ]'
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
  - name: test-pod
    image: quay.io/openshifttest/hello-
sdn@sha256:c89445416459e7adea9a5a416b3365ed3d74f2491beb904d61dc8d1eb89a72a4

    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop: [ALL]
```

where:

**k8s.v1.cni.cncf.io/networks,interface**

Specifies the name to be used as the **master** for the IPVLAN interface.

9. Apply the YAML file by running the following command:

```
$ oc apply -f pod-a.yaml
```

### Verification

1. Verify that the pod is running by using the following command:

```
$ oc get pod -n test-namespace
```

```
NAME    READY   STATUS    RESTARTS   AGE
pod-a   1/1     Running   0          2m36s
```

2. Show network interface information about the **pod-a** resource within the **test-namespace** by running the following command:

```
$ oc exec -n test-namespace pod-a -- ip a
```

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
3: eth0@if105: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1400 qdisc noqueue
state UP group default
    link/ether 0a:58:0a:d9:00:5d brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.217.0.93/23 brd 10.217.1.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::488b:91ff:fe84:a94b/64 scope link
       valid_lft forever preferred_lft forever
4: net1@if107: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether be:da:bd:7e:f4:37 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.0.0.2/24 brd 10.0.0.255 scope global net1
       valid_lft forever preferred_lft forever
    inet6 fe80::bcda:bdff:fe7e:f437/64 scope link
       valid_lft forever preferred_lft forever
5: net2@net1: <BROADCAST,MULTICAST,NOARP,UP,LOWER_UP> mtu 1500 qdisc
noqueue state UNKNOWN group default
    link/ether be:da:bd:7e:f4:37 brd ff:ff:ff:ff:ff:ff
    inet 10.0.0.1/24 brd 10.0.0.255 scope global net2
       valid_lft forever preferred_lft forever
    inet6 fe80::beda:bd00:17e:f437/64 scope link
       valid_lft forever preferred_lft forever
```

This output shows that the network interface **net2** associates with the physical interface **net1**.

## 4.9. REMOVING AN ADDITIONAL NETWORK

To clean up unused network configurations or free up network resources in OpenShift Container Platform, you can remove an additional network attachment. Delete the **NetworkAttachmentDefinition** custom resource to remove the secondary network from your cluster.

### 4.9.1. Removing a secondary NetworkAttachmentDefinition custom resource

To clean up unused network configurations or free up network resources in OpenShift Container Platform, you can remove a secondary **NetworkAttachmentDefinition** CR. Edit the Cluster Network Operator CR and delete the **NetworkAttachmentDefinition** CR to remove the secondary network from your cluster.

When a secondary network is removed from the cluster, it is not removed from any pods that it is attached to.

**Prerequisites**

- Install the OpenShift CLI (**oc**).
- Log in as a user with **cluster-admin** privileges.

**Procedure**

1. Edit the Cluster Network Operator (CNO) in your default text editor by running the following command:

```
$ oc edit networks.operator.openshift.io cluster
```

2. Modify the custom resource (CR) by removing the configuration that the CNO created from the **additionalNetworks** collection for the secondary network that you want to remove.

```
apiVersion: operator.openshift.io/v1
kind: Network
metadata:
  name: cluster
spec:
  additionalNetworks: []
```

where:

**spec.additionalNetworks**

Specifies the secondary network attachment definition that you want to remove from the **additionalNetworks** collection. If you are removing the configuration mapping for the only secondary network attachment definition in the **additionalNetworks** collection, you must specify an empty collection.

3. Remove the **NetworkAttachmentDefinition** CR from the network of your cluster by entering the following command:

```
$ oc delete net-attach-def <name_of_network_attachment_definition>
```

Replace **<name_of_network_attachment_definition>** with the name of the **NetworkAttachmentDefinition** CR that you want to remove.

4. Save your changes and quit the text editor to commit your changes.

5. Optional: Confirm that the secondary network CR was deleted by running the following command:

```
$ oc get network-attachment-definition --all-namespaces
```

## 4.10. ENABLING MULTI-NETWORKING FOR ADVANCED USE CASES WITH CNI PLUGIN CHAINING

You can use Container Network Interface (CNI) plugin chaining to enable advanced multi-networking use cases for your pods.

### 4.10.1. About CNI chaining

CNI plugin chaining allows pods to use multiple network interfaces. This enables advanced configurations such as traffic isolation and prioritized routing through granular traffic policies.

By using CNI plugin chaining, different types of traffic can be isolated to meet performance, security, and compliance requirements, providing greater flexibility in network design and traffic management.

Some scenarios where this might be useful include:

- **Multi-Network topologies**: Enables you to attach pods to multiple networks, each with its own traffic policy, where relevant.
- **Traffic isolation**: Provides separate networks for management, storage, and application traffic to ensure each has the appropriate security and QoS settings.
- **Custom routing rules**: Ensures that specific traffic, for example SIP traffic, always uses a designated network interface, while other traffic follows the default network.
- **Enhanced network performance**: Allows you to prioritize certain traffic types or manage congestion by directing them through dedicated network interfaces.

### 4.10.2. Configuring plugin chaining with the route-override CNI plugin

Plugin chaining allows you to configure multiple CNI plugins to be applied sequentially to the same network interface, where each plugin in the chain processes the interface in order.

When you define a **NetworkAttachmentDefinition** (NAD) with a **plugins** array, the first plugin can create the interface, and a second plugin can modify its routing configuration.

The **route-override** CNI plugin is commonly used as the second plugin in a chain to modify the routing configuration of an interface created by the first plugin. It supports the following operations:

- **addroutes:** Add static routes to direct traffic for specific destination networks through the interface.
- **delroutes:** Remove specific routes from the interface.
- **flushroutes:** Remove all routes from the interface.
- **flushgateway:** Remove the default gateway route from the interface.

The following example demonstrates plugin chaining by configuring a pod with two additional network interfaces, each on a separate VLAN with custom routing:

- **eth1** on the **192.168.100.0/24** network (VLAN 100), with a static route directing **10.0.0.0/8** traffic through this interface.
- **eth2** on the **192.168.200.0/24** network (VLAN 200), with a static route directing **172.16.0.0/12** traffic through this interface.

Each interface uses a chain of two plugins: **macvlan** to create the interface on a VLAN, and **route-override** to add static routes that direct specific traffic through that interface.

**Prerequisites**

- Install the OpenShift CLI (**oc**).
- An account with **cluster-admin** privileges.

**Procedure**

1. Create a namespace for the example by running the following command:
   -

```
$ oc create namespace chain-example
```

2. Create the first NetworkAttachmentDefinition (NAD) with a chained plugin configuration.

   a. Create a YAML file, such as **management.yaml**, to define a NAD that configures a new interface, **eth1**, on VLAN 100 with the following configuration:

```
apiVersion: k8s.cni.cncf.io/v1
kind: NetworkAttachmentDefinition
metadata:
  name: management-net
  namespace: chain-example
spec:
  config: '{
    "cniVersion": "1.0.0",
    "name": "management-net",
    "plugins": [
      {
        "type": "macvlan",
        "master": "br-ex",
        "vlan": 100,
        "mode": "bridge",
        "ipam": {
          "type": "static",
          "addresses": [
            {
              "address": "192.168.100.10/24",
              "gateway": "192.168.100.1"
            }
          ]
        }
      },
      {
        "type": "route-override",
        "addroutes": [
          {
            "dst": "10.0.0.0/8",
            "gw": "192.168.100.1"
          }
        ]
      }
    ]
  }'
```

3. Create the NAD by running the following command:

```
$ oc apply -f management.yaml
```

4. Create the second NAD with a chained plugin configuration.

   a. Create a YAML file, such as **sip.yaml**, to define a NAD that configures a new interface, **eth2**, on VLAN 200 with the following configuration:

```
apiVersion: k8s.cni.cncf.io/v1
kind: NetworkAttachmentDefinition
```

```
  metadata:
    name: sip-net
    namespace: chain-example
  spec:
    config: '{
      "cniVersion": "1.0.0",
      "name": "sip-net",
      "plugins": [
        {
          "type": "macvlan",
          "master": "br-ex",
          "vlan": 200,
          "mode": "bridge",
          "ipam": {
            "type": "static",
            "addresses": [
              {
                "address": "192.168.200.10/24",
                "gateway": "192.168.200.1"
              }
            ]
          }
        },
        {
          "type": "route-override",
          "addroutes": [
            {
              "dst": "172.16.0.0/12",
              "gw": "192.168.200.1"
            }
          ]
        }
      ]
    }'
```

5. Create the NAD by running the following command:

```
$ oc apply -f sip.yaml
```

6. Attach the **NetworkAttachmentDefinition** resources to a pod by creating a pod definition file, such as **pod.yaml**, with the following configuration:

```
apiVersion: v1
kind: Pod
metadata:
  name: chain-test-pod
  namespace: chain-example
  labels:
    app: chain-test
  annotations:
    k8s.v1.cni.cncf.io/networks: '[
      { "name": "management-net", "interface": "eth1" },
      { "name": "sip-net", "interface": "eth2" }
    ]'
spec:
```

```
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
  - name: test-container
    image: registry.access.redhat.com/ubi9/ubi:latest
    command: ["sleep", "infinity"]
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop: ["ALL"]
```

7. Create the pod by running the following command:

```
$ oc apply -f pod.yaml
```

8. Verify the pod is running with the following command:

```
$ oc wait --for=condition=Ready pod/chain-test-pod -n chain-example --timeout=120s
```

Example output:

```
pod/chain-test-pod condition met
```

### Verification

1. Run the following command to list all network interfaces and their assigned IP addresses inside the pod. This verifies that the pod has the additional interfaces configured by plugin chaining:

```
$ oc exec chain-test-pod -n chain-example -- ip a
```

Example output:

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0@if31: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 8901 qdisc noqueue state
UP
    link/ether 0a:58:0a:83:02:19 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.131.2.25/23 brd 10.131.3.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::858:aff:fe83:219/64 scope link
       valid_lft forever preferred_lft forever
3: eth1@if5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc noqueue state
UP qlen 1000
    link/ether aa:25:73:ff:a7:00 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.100.10/24 brd 192.168.100.255 scope global eth1
       valid_lft forever preferred_lft forever
    inet6 fe80::a825:73ff:feff:a700/64 scope link
       valid_lft forever preferred_lft forever
```

```
4: eth2@if5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc noqueue state
UP qlen 1000
    link/ether aa:a4:6c:4e:e8:97 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.200.10/24 brd 192.168.200.255 scope global eth2
       valid_lft forever preferred_lft forever
    inet6 fe80::a8a4:6cff:fe4e:e897/64 scope link
       valid_lft forever preferred_lft forever
```

This output shows the pod has three network interfaces:

- **eth0:** The default interface, connected to the cluster network.
- **eth1:** The first additional interface from **management-net**, with IP **192.168.100.10**.
- **eth2:** The second additional interface from **sip-net**, with IP **192.168.200.10**.

2. Run the following command to verify that the **route-override** plugin added the expected static routes:

```
$ oc exec chain-test-pod -n chain-example -- ip route
```

Example output:

```
default via 10.132.0.1 dev eth0
10.0.0.0/8 via 192.168.100.1 dev eth1
10.132.0.0/23 dev eth0 proto kernel scope link src 10.132.1.97
10.132.0.0/14 via 10.132.0.1 dev eth0
100.64.0.0/16 via 10.132.0.1 dev eth0
169.254.0.5 via 10.132.0.1 dev eth0
172.16.0.0/12 via 192.168.200.1 dev eth2
172.30.0.0/16 via 10.132.0.1 dev eth0
192.168.100.0/24 dev eth1 proto kernel scope link src 192.168.100.10
192.168.200.0/24 dev eth2 proto kernel scope link src 192.168.200.10
```

This output confirms that the **route-override** plugin in each chain added the expected static routes:

- For **10.0.0.0/8 via 192.168.100.1 dev eth1**, traffic destined for **10.0.0.0/8** is routed through **eth1** via the **management-net** gateway. This route was added by the **route-override** plugin in the **management-net** chain.
- For **172.16.0.0/12 via 192.168.200.1 dev eth2**, traffic destined for **172.16.0.0/12** is routed through **eth2** via the **sip-net** gateway. This route was added by the **route-override** plugin in the **sip-net** chain.
- The connected subnet routes (**192.168.100.0/24** and **192.168.200.0/24**) were created by the **macvlan** plugin, while the default route uses **eth0**, the cluster network interface.

# CHAPTER 5. VIRTUAL ROUTING AND FORWARDING

## 5.1. ABOUT VIRTUAL ROUTING AND FORWARDING

You can use virtual routing and forwarding (VRF) to provide multi-tenancy functionality. For example, where each tenant has its own unique routing tables and requires different default gateways.

VRF reduces the number of permissions needed by cloud-native network function (CNF), and provides increased visibility of the network topology of secondary networks. VRF devices combined with IP address rules provide the ability to create virtual routing and forwarding domains.

Processes can bind a socket to the VRF device. Packets through the binded socket use the routing table associated with the VRF device. An important feature of VRF is that it impacts only OSI model layer 3 traffic and above so L2 tools, such as LLDP, are not affected. This allows higher priority IP address rules such as policy-based routing to take precedence over the VRF device rules directing specific traffic.

### 5.1.1. Benefits of secondary networks for pods for telecommunications operators

You can connect network functions to different customers' infrastructure by using the same IP address with the Container Network Interface (CNI) virtual routing and forwarding (VRF) plugin. Using the CNI VRF plugin keeps different customers isolated.

In telecommunications use cases, each CNF can potentially be connected to many different networks sharing the same address space. These secondary networks can potentially conflict with the cluster's main network CIDR.

With the CNI VRF plugin, IP addresses are overlapped with the OpenShift Container Platform IP address space. The CNI VRF plugin also reduces the number of permissions needed by CNF and increases the visibility of the network topologies of secondary networks.

# CHAPTER 6. ASSIGNING A SECONDARY NETWORK TO A VRF

As a cluster administrator, you can configure a secondary network for a virtual routing and forwarding (VRF) domain by using the CNI VRF plugin. The virtual network that this plugin creates is associated with the physical interface that you specify.

Using a secondary network with a VRF instance has the following advantages:

**Workload isolation**

Isolate workload traffic by configuring a VRF instance for the secondary network.

**Improved security**

Enable improved security through isolated network paths in the VRF domain.

**Multi-tenancy support**

Support multi-tenancy through network segmentation with a unique routing table in the VRF domain for each tenant.

NOTE

Applications that use VRFs must bind to a specific device. The common usage is to use the **SO_BINDTODEVICE** option for a socket. The **SO_BINDTODEVICE** option binds the socket to the device that is specified in the passed interface name, for example, **eth1**. To use the **SO_BINDTODEVICE** option, the application must have **CAP_NET_RAW** capabilities.

Using a VRF through the **ip vrf exec** command is not supported in OpenShift Container Platform pods. To use VRF, bind applications directly to the VRF interface.

**Additional resources**

- About virtual routing and forwarding

## 6.1. CREATING A SECONDARY NETWORK ATTACHMENT WITH THE CNI VRF PLUGIN

The Cluster Network Operator (CNO) manages secondary network definitions. When you specify a secondary network in the cluster-scoped **Network** custom resource (CR), the CNO automatically creates the **NetworkAttachmentDefinition** CR.

NOTE

Do not edit the **NetworkAttachmentDefinition** CRs that the Cluster Network Operator manages. Doing so might disrupt network traffic on your secondary network.

**Prerequisites**

- Install the OpenShift CLI (**oc**).
- Log in to the cluster as a user with **cluster-admin** privileges.

**Procedure**

1. Create the **Network** CR for the additional network attachment and insert the **rawCNIConfig** configuration for the secondary network. Save as the **additional-network-attachment.yaml** file.

```
apiVersion: operator.openshift.io/v1
kind: Network
metadata:
  name: cluster
spec:
  additionalNetworks:
    - name: test-network-1
      namespace: additional-network-1
      type: Raw
      rawCNIConfig: '{
        "cniVersion": "0.3.1",
        "name": "macvlan-vrf",
        "plugins": [
        {
          "type": "macvlan",
          "master": "eth1",
          "ipam": {
              "type": "static",
              "addresses": [
              {
                  "address": "191.168.1.23/24"
              }
              ]
          }
        },
        {
          "type": "vrf",
          "vrfname": "vrf-1",
          "table": 1001
        }]
      }'
```

where:

**plugins**

You must specify a list. The first item in the list must be the secondary network underpinning the VRF network. The second item in the list is the VRF plugin configuration.

**type**

You must set this parameter to **vrf**.

**vrfname**

The name of the VRF that the interface is assigned to. If the VRF does not exist in the pod, the CNI creates the VRF.

**table**

Optional parameter. Specify the routing table ID. By default, the **tableid** parameter is used. If you do not specify a table ID, the CNI assigns a free routing table ID to the VRF.

NOTE

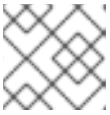
VRF functions correctly only when the resource is of type **netdevice**.

2. Create the **Network** resource:

```
$ oc create -f additional-network-attachment.yaml
```

3. Confirm that the CNO created the **NetworkAttachmentDefinition** CR by running the following command. Replace **<namespace>** with the namespace that you specified when configuring the network attachment, for example, **additional-network-1**. The expected output shows the name of the NAD CR and the creation age in minutes.

```
$ oc get network-attachment-definitions -n <namespace>
```



NOTE

A delay might exist before the CNO creates the CR.

### Verification

1. Create a pod and assign the pod to the secondary network that includes the VRF plugin configuration.

   a. Create a YAML file that defines the **Pod** resource, as demonstrated in the following **pod-additional-net.yaml** file:

```
apiVersion: v1
kind: Pod
metadata:
 name: pod-additional-net
 annotations:
   k8s.v1.cni.cncf.io/networks: '[
     {
           "name": "test-network-1" 1
     }
 ]'
spec:
 containers:
 - name: example-pod-1
   command: ["/bin/bash", "-c", "sleep 9000000"]
   image: centos:8
```

   where:

   **name**

   Specify the name of the secondary network that includes the VRF plugin configuration.

   b. Create the **Pod** resource by running the following command. The expected output shows the name of the **Pod** resource and the creation age in minutes.

```
$ oc create -f pod-additional-net.yaml
```

2. Verify that the pod network attachment connects to the VRF secondary network. Start a remote session with the pod and run the following command. The expected output shows the name of the VRF interface and its unique ID in the routing table.

```
$ ip vrf show
```

3. Confirm that the VRF interface is the controller for the secondary interface by entering the following command:

```
$ ip link
```

```
5: net1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master red
state UP mode
```